

OFFICIAL MICROSOFT LEARNING PRODUCT

2778A

**Writing Queries Using Microsoft® SQL
Server® 2008 Transact-SQL**

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

The names of manufacturers, products, or URLs are provided for informational purposes only and Microsoft makes no representations and warranties, either expressed, implied, or statutory, regarding these manufacturers or the use of the products with any Microsoft technologies. The inclusion of a manufacturer or product does not imply endorsement of Microsoft of the manufacturer or product. Links may be provided to third party sites. Such sites are not under the control of Microsoft and Microsoft is not responsible for the contents of any linked site or any link contained in a linked site, or any changes or updates to such sites. Microsoft is not responsible for webcasting or any other form of transmission received from any linked site. Microsoft is providing these links to you only as a convenience, and the inclusion of any link does not imply endorsement of Microsoft of the site or the products contained therein.

© 2008 Microsoft Corporation. All rights reserved.

Microsoft, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are property of their respective owners.

Product Number: 2778A

Part Number: X17-38829

Released: 11/2008

MICROSOFT LICENSE TERMS

OFFICIAL MICROSOFT LEARNING PRODUCTS COURSEWARE – STUDENT EDITION – Pre-Release and Final Versions

These license terms are an agreement between Microsoft Corporation and you. Please read them. They apply to the licensed content named above, which includes the media on which you received it, if any. The terms also apply to any Microsoft

- updates,
- supplements,
- Internet-based services, and
- support services

for this licensed content, unless other terms accompany those items. If so, those terms apply.

By using the licensed content, you accept these terms. If you do not accept them, do not use the licensed content.

If you comply with these license terms, you have the rights below.

1. OVERVIEW.

Licensed Content. The licensed content includes software, printed materials, academic materials (online and electronic), and associated media.

License Model. The licensed content is licensed on a per copy per device basis.

2. INSTALLATION AND USE RIGHTS.

- a. **Licensed Device.** The licensed device is the device on which you use the licensed content. You may install and use one copy of the licensed content on the licensed device.
- b. **Portable Device.** You may install another copy on a portable device for use by the single primary user of the licensed device.
- c. **Separation of Components.** The components of the licensed content are licensed as a single unit. You may not separate the components and install them on different devices.
- d. **Third Party Programs.** The licensed content may contain third party programs. These license terms will apply to your use of those third party programs, unless other terms accompany those programs.

3. PRE-RELEASE VERSIONS. If the licensed content is a pre-release ("beta") version, in addition to the other provisions in this agreement, then these terms also apply:

- a. **Pre-Release Licensed Content.** This licensed content is a pre-release version. It may not contain the same information and/or work the way a final version of the licensed content will. We may change it for the final, commercial version. We also may not release a commercial version. You will clearly and conspicuously inform any Students who participate in an Authorized Training Session and any Trainers who provide training in such Authorized Training Sessions of the foregoing; and, that you or Microsoft are under no obligation to provide them with any further content, including but not limited to the final released version of the Licensed Content for the Course.
- b. **Feedback.** If you agree to give feedback about the licensed content to Microsoft, you give to Microsoft, without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft software, licensed content, or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its software or documentation to third parties because we include your feedback in them. These rights survive this agreement.
- c. **Confidential Information.** The licensed content, including any viewer, user interface, features and documentation that may be included with the licensed content, is confidential and proprietary to Microsoft and its suppliers.
 - i. **Use.** For five years after installation of the licensed content or its commercial release, whichever is first, you may not disclose confidential information to third parties. You may disclose confidential information only to your employees and consultants who need to know the information. You must have written agreements with them that protect the confidential information at least as much as this agreement.
 - ii. **Survival.** Your duty to protect confidential information survives this agreement.

- iii. **Exclusions.** You may disclose confidential information in response to a judicial or governmental order. You must first give written notice to Microsoft to allow it to seek a protective order or otherwise protect the information. Confidential information does not include information that
 - becomes publicly known through no wrongful act;
 - you received from a third party who did not breach confidentiality obligations to Microsoft or its suppliers; or
 - you developed independently.
- d. **Term.** The term of this agreement for pre-release versions is (i) the date which Microsoft informs you is the end date for using the beta version, or (ii) the commercial release of the final release version of the licensed content, whichever is first ("beta term").
- e. **Use.** You will cease using all copies of the beta version upon expiration or termination of the beta term, and will destroy all copies of same in the possession or under your control.
- f. **Copies.** Microsoft will inform Authorized Learning Centers if they may make copies of the beta version (in either print and/or CD version) and distribute such copies to Students and/or Trainers. If Microsoft allows to such distribution, you will follow any additional terms that Microsoft provides to you for such copies and distribution.

4. ADDITIONAL LICENSING REQUIREMENTS AND/OR USE RIGHTS.

- a. **Media Elements and Templates.** You may use images, clip art, animations, sounds, music, shapes, video clips and templates provided with the licensed content solely for your personal training use. If you wish to use these media elements or templates for any other purpose, go to www.microsoft.com/permission to learn whether that use is allowed.
- b. **Academic Materials.** If the licensed content contains academic materials (such as white papers, labs, tests, datasheets and FAQs), you may copy and use the academic materials. You may not make any modifications to the academic materials and you may not print any book (either electronic or print version) in its entirety. If you reproduce any academic materials, you agree that:
 - The use of the academic materials will be only for your personal reference or training use
 - You will not republish or post the academic materials on any network computer or broadcast in any media;
 - You will include the academic material's original copyright notice, or a copyright notice to Microsoft's benefit in the format provided below:

Form of Notice:

© 2010 Reprinted for personal reference use only with permission by Microsoft Corporation. All rights reserved.

Microsoft and Windows are either registered trademarks or trademarks of Microsoft Corporation in the US and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

- c. **Distributable Code.** The licensed content may contain code that you are permitted to distribute in programs you develop if you comply with the terms below.
 - i. **Right to Use and Distribute.** The code and text files listed below are "Distributable Code."
 - REDIST.TXT Files. You may copy and distribute the object code form of code listed in REDIST.TXT files.
 - Sample Code. You may modify, copy, and distribute the source and object code form of code marked as "sample."
 - Third Party Distribution. You may permit distributors of your programs to copy and distribute the Distributable Code as part of those programs.
 - ii. **Distribution Requirements.** For any Distributable Code you distribute, you must
 - add significant primary functionality to it in your programs;
 - require distributors and external end users to agree to terms that protect it at least as much as this agreement;
 - display your valid copyright notice on your programs; and
 - indemnify, defend, and hold harmless Microsoft from any claims, including attorneys' fees, related to the distribution or use of your programs.

iii. Distribution Restrictions. You may not

- alter any copyright, trademark or patent notice in the Distributable Code;
- use Microsoft's trademarks in your programs' names or in a way that suggests your programs come from or are endorsed by Microsoft;
- distribute Distributable Code to run on a platform other than the Windows platform;
- include Distributable Code in malicious, deceptive or unlawful programs; or
- modify or distribute the source code of any Distributable Code so that any part of it becomes subject to an Excluded License. An Excluded License is one that requires, as a condition of use, modification or distribution, that
 - the code be disclosed or distributed in source code form; or
 - others have the right to modify it.

5. INTERNET-BASED SERVICES. Microsoft may provide Internet-based services with the licensed content. It may change or cancel them at any time. You may not use these services in any way that could harm them or impair anyone else's use of them. You may not use the services to try to gain unauthorized access to any service, data, account or network by any means.

6. SCOPE OF LICENSE. The licensed content is licensed, not sold. This agreement only gives you some rights to use the licensed content. Microsoft reserves all other rights. Unless applicable law gives you more rights despite this limitation, you may use the licensed content only as expressly permitted in this agreement. In doing so, you must comply with any technical limitations in the licensed content that only allow you to use it in certain ways. You may not

- disclose the results of any benchmark tests of the licensed content to any third party without Microsoft's prior written approval;
- work around any technical limitations in the licensed content;
- reverse engineer, decompile or disassemble the licensed content, except and only to the extent that applicable law expressly permits, despite this limitation;
- make more copies of the licensed content than specified in this agreement or allowed by applicable law, despite this limitation;
- publish the licensed content for others to copy;
- transfer the licensed content marked as 'beta' or 'pre-release' to any third party;
- allow others to access or use the licensed content;
- rent, lease or lend the licensed content; or
- use the licensed content for commercial licensed content hosting services.
- Rights to access the server software that may be included with the Licensed Content, including the Virtual Hard Disks does not give you any right to implement Microsoft patents or other Microsoft intellectual property in software or devices that may access the server.

7. BACKUP COPY. You may make one backup copy of the licensed content. You may use it only to reinstall the licensed content.

8. TRANSFER TO ANOTHER DEVICE. You may uninstall the licensed content and install it on another device for your personal training use. You may not do so to share this license between devices.

9. TRANSFER TO A THIRD PARTY. You may not transfer those versions marked as 'beta' or 'pre-release' to a third party. For final versions, these terms apply: The first user of the licensed content may transfer it and this agreement directly to a third party. Before the transfer, that party must agree that this agreement applies to the transfer and use of the licensed content. The first user must uninstall the licensed content before transferring it separately from the device. The first user may not retain any copies.

10. EXPORT RESTRICTIONS. The licensed content is subject to United States export laws and regulations. You must comply with all domestic and international export laws and regulations that apply to the licensed content. These laws include restrictions on destinations, end users and end use. For additional information, see www.microsoft.com/exporting.

11. NOT FOR RESALE SOFTWARE/LICENSED CONTENT. You may not sell software or licensed content marked as "NFR" or "Not for Resale."

12. ACADEMIC EDITION. You must be a "Qualified Educational User" to use licensed content marked as "Academic Edition" or "AE." If you do not know whether you are a Qualified Educational User, visit www.microsoft.com/education or contact the Microsoft affiliate serving your country.

13. ENTIRE AGREEMENT. This agreement, and the terms for supplements, updates, Internet-based services and support services that you use, are the entire agreement for the licensed content and support services.

14. APPLICABLE LAW.

- a. **United States.** If you acquired the licensed content in the United States, Washington state law governs the interpretation of this agreement and applies to claims for breach of it, regardless of conflict of laws principles. The laws of the state where you live govern all other claims, including claims under state consumer protection laws, unfair competition laws, and in tort.
- b. **Outside the United States.** If you acquired the licensed content in any other country, the laws of that country apply.

15. LEGAL EFFECT. This agreement describes certain legal rights. You may have other rights under the laws of your country. You may also have rights with respect to the party from whom you acquired the licensed content. This agreement does not change your rights under the laws of your country if the laws of your country do not permit it to do so.

16. DISCLAIMER OF WARRANTY. THE LICENSED CONTENT IS LICENSED "AS-IS." YOU BEAR THE RISK OF USING IT. MICROSOFT GIVES NO EXPRESS WARRANTIES, GUARANTEES OR CONDITIONS. YOU MAY HAVE ADDITIONAL CONSUMER RIGHTS UNDER YOUR LOCAL LAWS WHICH THIS AGREEMENT CANNOT CHANGE. TO THE EXTENT PERMITTED UNDER YOUR LOCAL LAWS, MICROSOFT EXCLUDES THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.

17. LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. YOU CAN RECOVER FROM MICROSOFT AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO U.S. \$5.00. YOU CANNOT RECOVER ANY OTHER DAMAGES, INCLUDING CONSEQUENTIAL, LOST PROFITS, SPECIAL, INDIRECT OR INCIDENTAL DAMAGES.

This limitation applies to

- anything related to the licensed content, software, services, content (including code) on third party Internet sites, or third party programs; and
- claims for breach of contract, breach of warranty, guarantee or condition, strict liability, negligence, or other tort to the extent permitted by applicable law.

It also applies even if Microsoft knew or should have known about the possibility of the damages. The above limitation or exclusion may not apply to you because your country may not allow the exclusion or limitation of incidental, consequential or other damages.

Please note: As this licensed content is distributed in Quebec, Canada, some of the clauses in this agreement are provided below in French.

Remarque : Ce le contenu sous licence étant distribué au Québec, Canada, certaines des clauses dans ce contrat sont fournies ci-dessous en français.

EXONÉRATION DE GARANTIE. Le contenu sous licence visé par une licence est offert « tel quel ». Toute utilisation de ce contenu sous licence est à votre seule risque et péril. Microsoft n'accorde aucune autre garantie expresse. Vous pouvez bénéficier de droits additionnels en vertu du droit local sur la protection des consommateurs, que ce contrat ne peut modifier. La ou elles sont permises par le droit local, les garanties implicites de qualité marchande, d'adéquation à un usage particulier et d'absence de contrefaçon sont exclues.

LIMITATION DES DOMMAGES-INTÉRÊTS ET EXCLUSION DE RESPONSABILITÉ POUR LES DOMMAGES. Vous pouvez obtenir de Microsoft et de ses fournisseurs une indemnisation en cas de dommages directs uniquement à hauteur de 5,00 \$ US. Vous ne pouvez prétendre à aucune indemnisation pour les autres dommages, y compris les dommages spéciaux, indirects ou accessoires et pertes de bénéfices.

Cette limitation concerne:

- tout ce qui est relié au le contenu sous licence , aux services ou au contenu (y compris le code) figurant sur des sites Internet tiers ou dans des programmes tiers ; et
- les réclamations au titre de violation de contrat ou de garantie, ou au titre de responsabilité stricte, de négligence ou d'une autre faute dans la limite autorisée par la loi en vigueur.

Elle s'applique également, même si Microsoft connaît ou devrait connaître l'éventualité d'un tel dommage. Si votre pays n'autorise pas l'exclusion ou la limitation de responsabilité pour les dommages indirects, accessoires ou de quelque nature que ce soit, il se peut que la limitation ou l'exclusion ci-dessus ne s'appliquera pas à votre égard.

EFFET JURIDIQUE. Le présent contrat décrit certains droits juridiques. Vous pourriez avoir d'autres droits prévus par les lois de votre pays. Le présent contrat ne modifie pas les droits que vous confèrent les lois de votre pays si celles-ci ne le permettent pas.

Acknowledgements

Microsoft Learning would like to acknowledge and thank the following for their contribution towards developing this title. Their effort at various stages in the development has ensured that you have a good classroom experience.

Peter Lammers – Lead Developer

Peter Lammers joined Aeshen in 2002 as a Product Analyst, and he has been a Lead Product Analyst since 2005, working on Microsoft TechNet Content, Webcasts, White Papers, and Microsoft Learning Courses. Prior to that he has been a computer programmer and network technician with a 14-year background in troubleshooting, training, modifying and supporting a software application; network administration, troubleshooting, and server, desktop, and firewall support.

Seth Wolf – Content Developer

Seth Wolf has been working with computing technology for over 20 years. His background includes programming, database design, Web site design, network management, hardware troubleshooting, and user support. He remembers the good old days of dBase and Btrieve.

Jerry Knowles - Content Developer

Mr. Knowles joined Aeshen in 2008 as an Application Analyst. He has worked in Information Technology since 1989 as an instructor, application developer, SQL database administrator, and consultant.

Jason Shigley - Content Developer

Mr. Shigley joined Aeshen in 2006 as an application developer and has contributed on a variety of technical education and development projects. He has worked in Information Technology since 1995 as an application developer, database administrator and systems architect.

Sean Masters – Content Developer

Mr. Masters joined Aeshen in 2007. He has worked in SMB technical operations for nearly 10 years including 4 years as manager of information technology at a property management firm and 4 years as a private consultant to various legal and financial firms in the New England area.

David Brandon – Content Developer

Mr. Brandon has served as the Development Leader at Computer Automation Systems since 2001. He came to Computer Automation Systems after serving as a Technical Team Leader at Acxiom Corporation. He has worked in software development and technical management for over 14 years.

Karl Middlebrooks – Subject Matter Expert

Mr. Middlebrooks is a Product Analyst with Aeshen, and joined in 2004. He has over 20 years experience in IT and Operations management, network administration, and database administration.

Contents

Module 1: Getting Started with Databases and Transact-SQL in SQL Server 2008

Lesson 1: Overview of SQL Server 2008	1-3
Lesson 2: Overview of SQL Server Databases	1-8
Lesson 3: Overview and Syntax Elements of T-SQL	1-14
Lesson 4: Working with T-SQL Scripts	1-23
Lesson 5: Using T-SQL Querying Tools	1-27
Lab: Using SQL Server Management Studio and SQLCMD	1-34

Module 2: Querying and Filtering Data

Lesson 1: Using the SELECT Statement	2-3
Lesson 2: Filtering Data	2-7
Lesson 3: Working with NULL Values	2-16
Lesson 4: Formatting Result Sets	2-20
Lesson 5: Performance Considerations for Writing Queries	2-37
Lab: Performing Basic Queries	2-31

Module 3: Grouping and Summarizing Data

Lesson 1: Summarizing Data by Using Aggregate Functions	3-3
Lesson 2: Summarizing Grouped Data	3-9
Lesson 3: Ranking Grouped Data	3-19
Lesson 4: Creating Crosstab Queries	3-26
Lab: Grouping and Summarizing Data	3-31

Module 4: Joining Data from Multiple Tables

Lesson 1: Querying Multiple Tables by Using Joins	4-3
Lesson 2: Applying Joins for Typical Reporting Needs	4-11
Lesson 3: Combining and Limiting Result Sets	4-17
Lab: Joining Data from Multiple Tables	4-24

Module 5: Working with Subqueries

Lesson 1: Writing Basic Subqueries	5-3
Lesson 2: Writing Correlated Subqueries	5-10
Lesson 3: Comparing Subqueries with Joins and Temporary Tables	5-16
Lesson 4: Using Common Table Expressions	5-20
Lab: Working with Subqueries	5-25

Module 6: Modifying Data in Tables

Lesson 1: Inserting Data into Tables	6-3
Lesson 2: Deleting Data from Tables	6-10
Lesson 3: Update Data in Tables	6-17
Lesson 4: Overview of Transactions	6-23
Lab: Modifying Data	6-33

Module 7: Querying Metadata, XML, and Full-Text Indexes

Lesson 1: Querying Metadata	7-3
Lesson 2: Overview of XML	7-14
Lesson 3: Querying XML Data	7-20
Lesson 4: Overview of Full-Text Indexes	7-26
Lesson 5: Querying Full-Text Indexes	7-31
Lab: Querying Metadata, XML, and Full-Text Indexes	7-38

Module 8: Using Programming Objects for Data Retrieval

Lesson 1: Overview of Views	8-3
Lesson 2: Overview of User-Defined Functions	8-13
Lesson 3: Overview of Stored Procedures	8-21
Lesson 4: Overview of Triggers	8-27
Lesson 5: Writing Distributed Queries	8-33
Lab: Using Programming Objects for Data Retrieval	8-38

Module 9: Using Advanced Techniques

Lesson 1: Considerations for Querying Data	9-3
Lesson 2: Working with Data Types	9-11
Lesson 3: Cursors and Set-Based Queries	9-19
Lesson 4: Dynamic SQL	9-26
Lesson 5: Maintaining Query Files	9-31
Lab: Using Advanced Techniques	9-34

Lab Answer Keys

About This Course

This section provides you with a brief description of the course, audience, suggested prerequisites, and course objectives.

Course Description

This three-day instructor-led course provides students with the technical skills required to write basic Transact-SQL queries for Microsoft SQL Server 2008.

Audience

This course is intended for SQL Server database administrators, implementers, system engineers, and developers who are responsible for writing queries.

Student Prerequisites

This course requires that you meet the following prerequisites:

- Knowledge of data integrity concepts
- Core Windows Server skills
- Relational database design skills.
- Programming skills

Course Objectives

After completing this course, students will be able to:

- Describe the uses of and ways to execute the Transact-SQL language.
- Use querying tools.
- Write SELECT queries to retrieve data.
- Group and summarize data by using Transact-SQL.
- Join data from multiple tables.
- Write queries that retrieve and modify data by using subqueries.
- Modify data in tables.
- Query text fields with full-text search.
- Describe how to create programming objects.
- Use various techniques when working with complex queries.

Course Outline

This section provides an outline of the course:

Module 1, Getting Started with Databases and Transact-SQL in SQL Server 2008

This module covers how client/server architecture works, and the various database and business tasks that can be performed by using the components of SQL Server 2008. It also covers SQL Server database concepts such as relational databases, normalization, and database objects. As well as, how to use T-SQL to query databases and generate reports.

Module 2, Querying and Filtering Data

This module covers the basic Transact-SQL (T-SQL) statements that are used for writing queries, filtering data, and formatting result sets.

Module 3, Grouping and Summarizing Data

This module covers grouping and summarizing data when generating reports in Microsoft SQL Server 2008 by using aggregate functions and the COMPUTE clause.

Module 4, Joining Data from Multiple Tables

This module covers writing joins to query multiple tables, as well as limiting and combining result sets.

Module 5, Working with Subqueries

This module covers basic and correlated subqueries and how these compare with joins and temporary tables. It also covers using common table expressions in queries.

Module 6, Modifying Data in Tables

This module covers modifying the data in tables by using the INSERT, DELETE, and UPDATE statements. In addition, it covers transactions work in a database, the importance of transaction isolation levels, and how to manage transactions.

Module 7, Querying Metadata, XML, and Full-Text Indexes

This module covers querying semi-structured and unstructured data. It also covers how SQL Server 2008 handles XML data and will query XML data and full-text indexing in SQL Server 2008.

Module 8, Using Programming Objects for Data Retrieval

This module covers user-defined functions and executing various kinds of queries by using user-defined functions. It also covers SQL Server views that encapsulate data and present users with limited and relevant information. In addition, it covers SQL Server stored procedures and the functionalities of the various programming objects. Also, how to perform distributed queries and how SQL Server works with heterogeneous data such as databases, spreadsheets, and other servers.

Module 9, Using Advanced Querying Techniques

This module covers best practices for querying complex data. It also covers how to query complex table structures such as data stored in hierarchies and self-referencing tables. It also covers the recommended guidelines for executing queries and how to optimize query performance.

Course Materials

The following materials are included with your kit:

- **Course Handbook** A succinct classroom learning guide that provides all the critical technical information in a crisp, tightly-focused format, which is just right for an effective in-class learning experience.
- **Lessons:** Guide you through the learning objectives and provide the key points that are critical to the success of the in-class learning experience.
- **Labs:** Provide a real-world, hands-on platform for you to apply the knowledge and skills learned in the module.
- **Module Reviews and Takeaways:** Provide improved on-the-job reference material to boost knowledge and skills retention.
- **Lab Answer Keys:** Provide step-by-step lab solution guidance at your finger tips when it's needed.



Course Companion Content on the [http://www.microsoft.com/learning/companionmoc/ Site:](http://www.microsoft.com/learning/companionmoc/)

Searchable, easy-to-navigate digital content with integrated premium on-line resources designed to supplement the Course Handbook.

- **Modules:** Include companion content, such as questions and answers, detailed demo steps and additional reading links, for each lesson. Additionally, they include Lab Review questions and answers and Module Reviews and Takeaways sections, which contain the review questions and answers, best practices, common issues and troubleshooting tips with answers, and real-world issues and scenarios with answers.
- **Resources:** Include well-categorized additional resources that give you immediate access to the most up-to-date premium content on TechNet, MSDN®, Microsoft Press®



Student Course files on the [http://www.microsoft.com/learning/companionmoc/ Site:](http://www.microsoft.com/learning/companionmoc/) Includes the

Allfiles.exe, a self-extracting executable file that contains all the files required for the labs and demonstrations.

- **Course evaluation** At the end of the course, you will have the opportunity to complete an online evaluation to provide feedback on the course, training facility, and instructor.
 - To provide additional comments or feedback on the course, send e-mail to support@mscourseware.com. To inquire about the Microsoft Certification Program, send e-mail to mchelp@microsoft.com.

Virtual Machine Environment

This section provides the information for setting up the classroom environment to support the business scenario of the course.

Virtual Machine Configuration

In this course, you will use Microsoft Virtual Server 2005 R2 with SP1 to perform the labs.

Important: At the end of each lab, you must close the virtual machine and must not save any changes. To close a virtual machine without saving the changes, perform the following steps: 1. On the virtual machine, on the **Action** menu, click **Close**. 2. In the **Close** dialog box, in the **What do you want the virtual machine to do?** list, click **Turn off and delete changes**, and then click **OK**.

The following table shows the role of each virtual machine used in this course:

Virtual machine	Role
2778A-NY-SQL-01	Windows Server 2008 with SQL Server 2008

Software Configuration

The following software is installed on each VM:

- Windows Server 2008 Enterprise Edition
- SQL Server 2008

Course Files

There are files associated with the labs in this course. The lab files are located in the folder E:\Labfiles on the student computers.

Classroom Setup

Each classroom computer will have the same virtual machine configured in the same way.

Course Hardware Level

To ensure a satisfactory student experience, Microsoft Learning requires a minimum equipment configuration for trainer and student computers in all Microsoft Certified Partner for Learning Solutions (CPLS) classrooms in which Official Microsoft Learning Product courseware are taught.

This course requires that you have a computer that meets or exceeds hardware level 5.5, which specifies a 2.4-gigahertz (GHz) (minimum) Pentium 4 or equivalent CPU, at least 2 gigabytes (GB) of RAM, 16 megabytes (MB) of video RAM, and two 7200 RPM 40-GB hard disks.

Module 1

Getting Started with Databases and Transact-SQL in SQL Server 2008

Contents:

Lesson 1: Overview of SQL Server 2008	1-3
Lesson 2: Overview of SQL Server Databases	1-8
Lesson 3: Overview and Syntax Elements of T-SQL	1-14
Lesson 4: Working with T-SQL Scripts	1-23
Lesson 5: Using T-SQL Querying Tools	1-27
Lab: Using SQL Server Management Studio and SQLCMD	1-34

Module Overview

- Overview of SQL Server 2008
- Overview of SQL Server Databases
- Overview and Syntax Elements of T-SQL
- Working with T-SQL Scripts
- Using T-SQL Querying Tools

Transact SQL or T-SQL is the primary programming language of Microsoft® SQL Server® 2008. A comprehensive understanding of T-SQL commands and the T-SQL language is important for all SQL Server Administrators. T-SQL commands can be used for querying a SQL Server. A query is a request for data that is stored in SQL Server.

By understanding how SQL Server 2008 utilizes TSQL and how to apply it to business needs, the administrator can be more responsive and comprehensive in design and troubleshooting. By understanding the underlying design of SQL Server 2008, the database administrator will be better prepared for new projects and challenges.

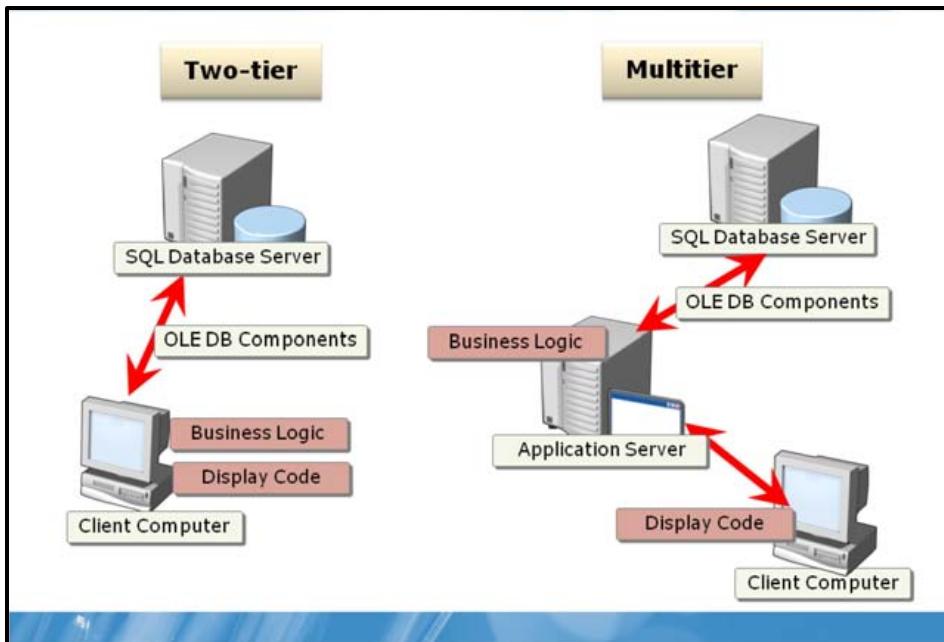
Lesson 1

Overview of SQL Server 2008

- Overview of Client/Server Architecture
- SQL Server Components
- SQL Server Management Tools
- SQL Server Database Engine Components

It is important to understand the basic underlying principles of SQL Server. The SQL Server 2008 Database Engine is a service for storing and processing data in either a relational format or as XML documents. Understanding these components and their related tools is fundamental to building and managing SQL Server systems.

Overview of Client/Server Architecture



Key Points

Client/Server Architecture is the use of a client to connect to a server for processing.

- In a two-tier client/server system, users run an application on their local computer, known as a client, which connects over a network to the server running SQL Server.
- The client application runs both business logic and the code to display output to the user, and is also known as a thick client.
- You can also use SQL Server in a Client/Server Application where the databases are on a server and you use an application that connects for backend process.

Question: What types of Client/Server architecture systems do you use at your current organization?

SQL Server Components

Server Components	Description
SQL Server Database Engine	Core service for storing and processing data
Analysis Services	Tools for creating and managing analytical processing
Reporting Services	Components for creating and deploying reports
Integration Services	Tools for moving, copying, and transforming data
The Database Engine also features these components:	
• Full-Text Search	• Replication
• Service Broker	• Notification Services

Key Points

There are many optional components of SQL Server.

- Use the Feature Selection page of the **SQL Server Installation Wizard** to select the components to include in an installation of SQL Server.
- By default, none of the features in the tree are selected.

Question: Have you used any of these SQL Server components before?

SQL Server Management Tools

Management tools	Description
 SQL Server Management Studio	An environment to access, configure, manage, and administer SQL components
 SQL Server Configuration Manager	An interface to provide management for SQL services, protocols, and client aliases
 SQL Server Profiler	A GUI tool to profile and trace the Database Engine and Analysis Services
 Database Engine Tuning Advisor	An application to create an optimal sets of indexes, indexed views, and partitions
 Business Intelligence Development Studio	An IDE for creating Analysis Services, Reporting Services, and Integration Services solutions

Key Points

There are several important management tools in SQL Server.

- Use the Feature Selection page of the **SQL Server Installation Wizard** to select the SQL Server Management Tools to include in an installation of SQL Server.
- The Connectivity Components installs components for communication between clients and servers, and network libraries for DB-Library, ODBC, and OLE DB.

Question: Have you used any of these SQL Server management tools before?

SQL Server Database Engine Components

Components	Description
• Protocols	Ways to implement the external interface to the SQL Server
• Relational Engine	Interface into the storage engine, composed of services to interact with the underlying database storage components and features
• Storage Engine	Core of SQL Server, a highly scalable and available service for data storage, processing, and security
• SQLOS	Operating system with a powerful API, which brings together all system components, enabling innovation of SQL Server's scalability and performance, providing manageability and supportability features

Key Points

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.
- Use the Database Engine to create relational databases for online transaction processing or online analytical processing data.
- The latest version of SQL Server features SQLOS—a user level highly configurable operating system with powerful API, enabling automatic locality and advanced parallelism.
- SQLOS provides operating system services such as a non-preemptive scheduling, memory management, deadlock detection, exception handling, hosting for external components such as Common Language Runtime (CLR) and other services.
- SQLOS exposes cohesive API to developers so that they can easily exploit features of hardware and the operating system.
- The SOS scheduling subsystem consists of scheduling nodes, schedulers, tasks, workers and system threads.
- SQLOS memory management consists of memory nodes, memory clerks, caches, pools, and memory objects.
- SQLOS contains several threads to provide self monitoring and adequate resource distribution.

Lesson 2

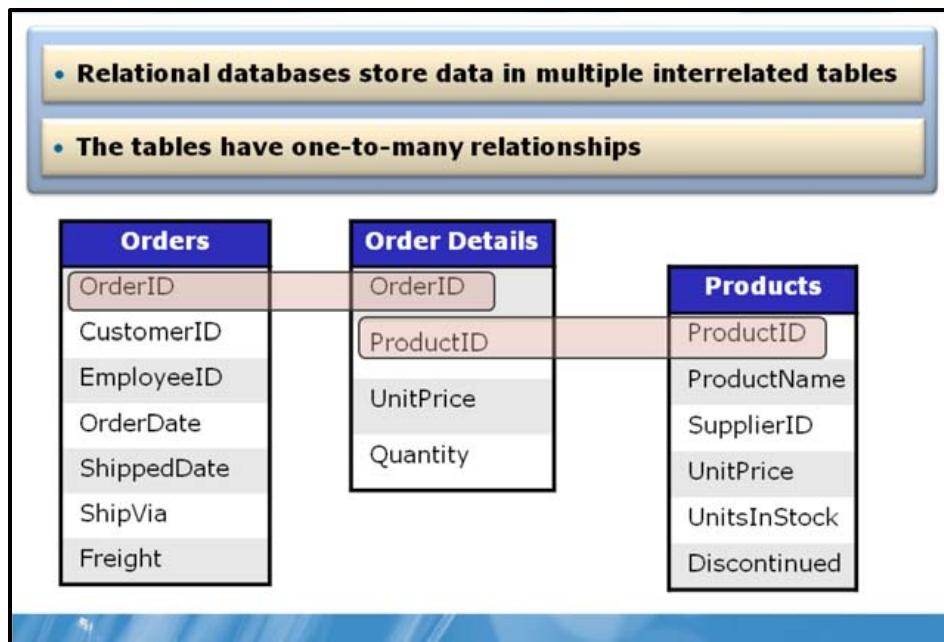
Overview of SQL Server Databases

- Overview of Relational Databases
- What Is Normalization?
- The Normalization Process
- SQL Server Database Objects
- Overview of Data Types

Relational databases are powerful tools. Behind just about any complex application is a relational database. The tables, views, indexes, and stored procedures that constitute a relational database are the gears of these applications.

Understanding the basic principles of relational databases, database normalization, and data types provides administrators with the knowledge to build powerful, highly scalable, reliable SQL Server enterprises.

Overview of Relational Databases



Key Points

A relational database is a complex database that stores data in multiple tables that are interrelated.

- Usually the tables in a relational database have one-to-many relationships.
- The relational database server of SQL Server has two main parts: the relational engine and the storage engine.
- The two engines work independently, interacting with each other through native data access components such as Object Linking and Embedding and Database (OLE DB).

What Is Normalization?

The process for removing redundant data from a database

Benefits

- ✓ Accelerates sorting and indexing
- ✓ Allows more clustered indexes
- ✓ Helps UPDATE performance
- ✓ More compact databases

Disadvantages

- Increase in tables to join
- Slower data retrieval
- Insertion of code in tables
- Difficulty in data model query

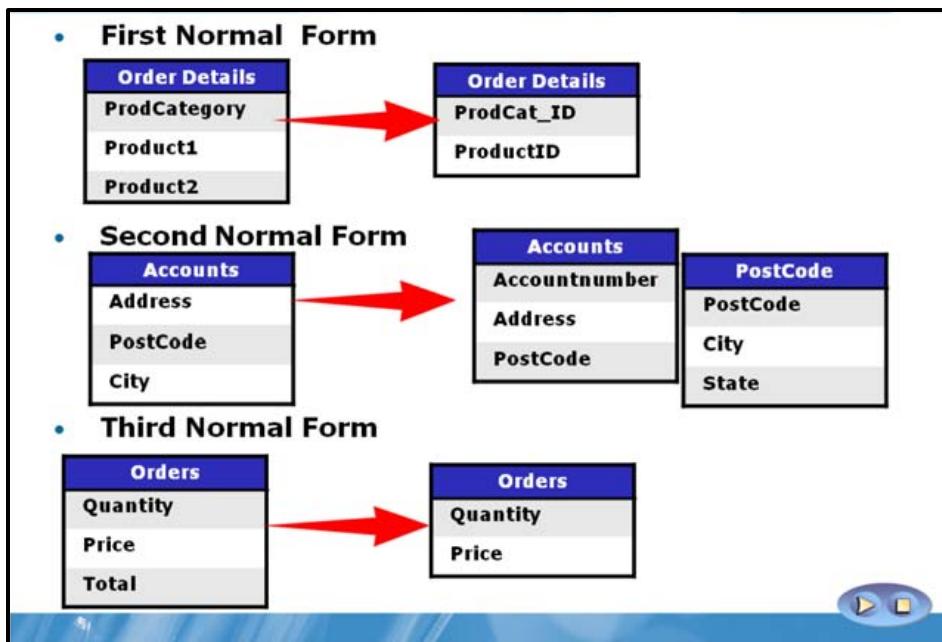
Key Points

Normalization is the process of organizing data in a database.

- The normalization process includes creating tables and establishing relationships between those tables according to rules designed to protect the data and to make the database more flexible by eliminating redundancy.
- Reasonable normalization of a database design yields the best performance.

Question: Have you ever implemented normalization before? Did you find that it improved SQL Server performance?

The Normalization Process



Key Points

The normalization process includes the first, second, and third normal form.

- With SQL Server, reasonable normalization often helps rather than hurts performance.
- As normalization increases, so do the number and complexity of joins required to retrieve data.
- There are a few rules for database normalization. Each rule is called a "normal" form. If the first rule is observed, the database is said to be in "first normal form". If the first three rules are observed, the database is considered to be in "third normal form."
- Although other levels of normalization are possible, third normal form is considered the highest level necessary for most applications.

SQL Server Database Objects

Objects	Notes
Tables	Contain all the data in SQL Server databases
Views	Act like a virtual table or a stored query
Indexes	Enable fast retrieval, built from one or more columns in table or view
Triggers	Execute a batch of SQL code when an insert, update or delete command is executed against a specific table
Procedures	Accept parameters, contain statements, and return values.
Constraints	Prevent inconsistent data from being placed in a column
Rules	Specify acceptable values that can be inserted in column

Key Points

There are many different database objects.

- Tables are the main form for collection of information. Tables are objects that contain all the data in SQL Server databases.
- A view can be thought of as either a virtual table or a stored query. The data accessible through a view is not stored in the database as a distinct object. What is stored in the database is a SELECT statement.
- An index is an on-disk structure associated with a table or view that speeds retrieval of rows from the table or view.
- The main difference between a trigger and a stored procedure is that the former is attached to a table and is only fired when an INSERT, UPDATE or DELETE occurs.
- Stored procedures in 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.
- The primary job of a constraint is to enforce a rule in the database.

Overview of Data Types

Each column, variable, expression, parameter has a data type

**A data type specifies the type of data the object can hold:
integers, characters, monetary data, date and time, binary**

Data types are organized into the following categories:

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



Key Points

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 T-SQL or the Microsoft® .NET Framework.
- Alias data types are based on the system-supplied data types.
- User-defined types obtain their characteristics from the methods and operators of a class that you create.

Lesson 3

Overview and Syntax Elements of T-SQL

- A History and Definition of ANSI SQL and T-SQL
- Categories of SQL Statements
- Introduction to Basic T-SQL Syntax
- Types of T-SQL Operators
- What Are T-SQL Functions?
- What Are T-SQL Variables?
- What Are T-SQL Expressions?
- Control-of-flow Statements

It is important to not only understand how to use and write T-SQL code, but also understand the organizational types of T-SQL commands. Comprehensive knowledge of T-SQL allows administrators to manage SQL Server systems and the code that operates them.

A History and Definition of ANSI SQL and T-SQL

- Developed in the early 1970s
- ANSI-SQL defined by the American National Standards Institute
- Microsoft implementation is T-SQL, or Transact SQL
- Other implementations include PL/SQL and SQL Procedural Language.

Key Points

SQL was developed by IBM in the early 1970s. SQL was adopted as a standard by the American National Standards Institute (ANSI) in 1986 and the International Standards Organization (ISO) in 1987.

- Microsoft's implementation of SQL is known as T-SQL. It is the language that is used to create queries for SQL Server.
- There are other implementations of SQL such as Oracle's PL/SQL, Procedural Language/SQL and IBM's SQL Procedural Language.

Question: Have you used any other implementation of SQL besides T-SQL?

Categories of SQL Statements

- DML – Data Manipulation Language
 - INSERT Table1 VALUES (1, 'Row #1')
- DCL – Data Control Language
 - GRANT CONNECT TO guest;
 - GRANT CONTROL SERVER TO user_name;
- DDL – Data Definition Language
 - CREATE USER user_name
- TCL - Transactional Control Language
 - COMMIT TRANSACTION Inner2;
- DQL - SQL Select Statements
 - SELECT ProductID, Name, ListPrice
FROM Production.Product



Key Points

There are five categories of SQL statements.

- DML is the category of SQL statements that included changes to the data within the database. These include the UPDATE, DELETE, and INSERT statements.
- DCL is the category of SQL statements that are associated with rights to objects within the database. These include GRANT, REVOKE, and DENY.
- DDL is the category of SQL statements that are associated with the implementation, changing, or deletion of objects for or within a database. These include CREATE, TRUNCATE, DROP, and ALTER.
- TCL is abbreviation of Transactional Control Language. It is used to manage different transactions occurring within a database. These include COMMIT, SAVE POINT, ROLLBACK.
- SQL Queries are often considered part of the DML. But when started with a SELECT, there are no changes to the data used in the query.

Introduction to Basic T-SQL Syntax

There are four primary properties to the SELECT statement

- 1 The number and attributes of the columns in the result set**
- 2 The tables from which the result set data is retrieved**
- 3 The conditions the rows in the source tables must meet**
- 4 The sequence which the rows of the result set are ordered**

```
SELECT ProductID, Name, ListPrice
FROM Production.Product
WHERE ListPrice > $40
ORDER BY ListPrice ASC
```



Key Points

There are four primary properties to the SELECT statement.

- The SELECT statement retrieves data from SQL Server and returns it to the user in one or more result sets.
- A result set is a tabular arrangement of the data from the SELECT. Like an SQL table, the result set is made up of columns and rows.

Types of T-SQL Operators

Type	Operators
• Arithmetic operators	• +, -, *, /, % <i>Vacation + SickLeave AS 'Total PTO'</i>
• Assignment operator	• = <i>SET @MyCounter = 1</i>
• Comparison operators	• =, <, >, <>, !, >=, <=
• Logical operators	• AND, OR, NOT <i>WHERE Department = 'Sales' AND (Shift = 'Evening' OR Shift = 'Night')</i>
• String concatenation operator	• + <i>SELECT LastName + ', ' + FirstName AS Moniker</i>

Key Points

Operators provide various ways of manipulating and comparing information.

- Logical Operators are used to specify how multiple search terms are combined in a search query.
- Logical operators also create more complicated search expressions from simpler ones, and thus refine your search.

What Are T-SQL Functions?

Functions	Notes
Rowset	<ul style="list-style-type: none">• Return objects that can be used as table references
Examples:	CONTAINSTABLE, OPENDATASOURCE, OPENQUERY
Aggregate	<ul style="list-style-type: none">• Operate on a collection but returns a single value
Examples:	AVG, CHECKSUM_AGG, SUM, COUNT
Ranking	<ul style="list-style-type: none">• Return a ranking value for each row in a partition
Examples:	RANK, DENSE_RANK
Scalar	<ul style="list-style-type: none">• Operate on a single value and then return a single value
Examples:	CREATE FUNCTION dbo.ufn_CubicVolume

Key Points

The above list is a sample of what is available in T-SQL.

- Additional user-defined functions are also available.
- Scalar functions can be used wherever an expression is valid.

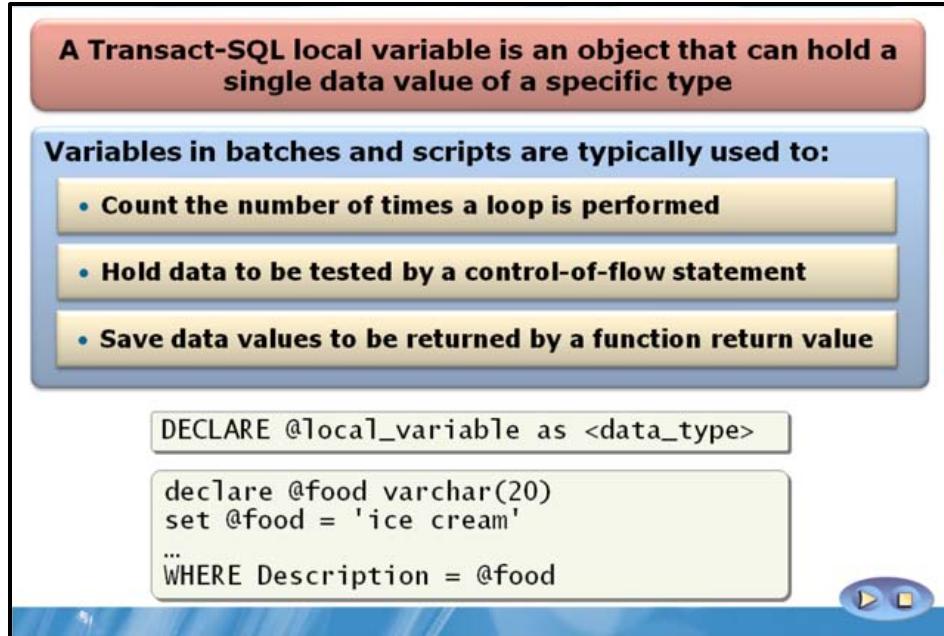
What Are T-SQL Variables?

A Transact-SQL local variable is an object that can hold a single data value of a specific type

Variables in batches and scripts are typically used to:

- Count the number of times a loop is performed
- Hold data to be tested by a control-of-flow statement
- Save data values to be returned by a function return value

```
DECLARE @local_variable as <data_type>
declare @food varchar(20)
set @food = 'ice cream'
...
WHERE Description = @food
```



Key Points

A Transact-SQL local variable is an object that can hold a single data value of a specific type.

- The variable has a data type that is defined.
- When a variable is first declared, its value is set to NULL.
- To assign a value to a variable, use the SET statement.
- A variable can also have a value assigned by being referenced in the select list of a SELECT statement.

What Are T-SQL Expressions?

Symbols and operators evaluated to obtain a single value

Expressions can be combined if one of these is true:

- The expressions have the same data type
- The data type with the lower precedence can be converted to the data type with the higher precedence

```
SELECT ProductID, Variable_N +2
```

Key Points

Expressions are combination of symbols and operators that the SQL Server Database Engine evaluates to obtain a single data value.

- Simple expressions can be a single constant, variable, column, or scalar function.
- Operators can be used to join two or more simple expressions into a complex expression.

Control-of-flow Statements

These are the control-of-flow keywords:

- BEGIN...END
- BREAK
- GOTO
- CONTINUE
- IF...ELSE
- WHILE
- RETURN
- WAITFOR

```
IF Boolean_expression
BEGIN
    { sql_statement | statement_block }
END
ELSE
    { sql_statement | statement_block }
```

Key Points

T-SQL provides special words called control-of-flow language that control the flow of execution of T-SQL statements, statement blocks, user-defined functions, and stored procedures.

- Without control-of-flow language, separate T-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 T-SQL to take some kind of action.

Lesson 4

Working with T-SQL Scripts

- What Are Batch Directives?
- Structured Exception Handling
- Commenting T-SQL Code

Writing T-SQL scripts is easy, but it is important to understand the basic principles, like commenting code in order to write scripts that will have solid underlying structure.

What Are Batch Directives?

- These control movement within a T-SQL file

```
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)))+'.
GO
```

Key Points

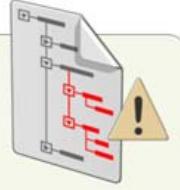
Batch directives control movement within a T-SQL file.

- Batch directives impose conditions on the execution of T-SQL statements.
- GO is a batch directive which signals to SQL Server to execute the batch.

Structured Exception Handling

TRY/CATCH

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



RAISERROR

```
RAISERROR (N'This is message %s %d.', -- Message text.
           10, -- Severity,
           1, -- State,
           N'number', -- First argument.
           5); -- Second argument.
-- The message text returned is: This is message number 5.
GO
```

Key Points

TRY...CATCH implements error handling for T-SQL that is similar to the exception handling in the Microsoft® Visual C#® and Microsoft® Visual C++® languages.

- A group of SQL statements are enclosed within a TRY statement. If an error is found control is passed to the CATCH block. The CATCH block follows immediately after the TRY block.
- RAISERROR allows the creation of an error message. These can be used within a TRY/CATCH to create an event that displays a customized message.

Commenting T-SQL Code

- **Comments are statements about the meaning of the code**
- **When used, there is no execution performed on the text**

There are two ways to comment code using T-SQL:

- **The use of a beginning /* and ending */ creates comments**

```
/*
This is a comment
*/
```

- **The double dash comments to the end of line**

```
--This is a comment
```

Key Points

Comments are non-executing text strings in program code (also known as remarks).

- Comments can be used to document code or temporarily disable parts of T-SQL statements and batches being diagnosed.
- Using comments to document code makes future program code maintenance easier.
- Comments are often used to record the program name, the author name, and the dates of major code changes.
- Comments can be used to describe complicated calculations or explain a programming method.

Lesson 5

Using T-SQL Querying Tools

- Tools for Querying SQL Server 2008 Databases
- An Introduction to SQL Server Management Studio
- What Is a SQL Server Solution?
- Creating SQL Server Solutions
- Executing Queries in SQL Server Management Studio
- Generating Reports in Microsoft Office Excel

There are many tools available for creating and maintaining SQL Server T-SQL code structures. It is important to understand how T-SQL scripts are grouped and organized into projects.

Tools for Querying SQL Server 2008 Databases

Tool	Description
• SQL Server Management Studio	<ul style="list-style-type: none">Used for interactive creation of T-SQL scriptsTo access, configure, manage, and create many other SQL Server Objects
• Microsoft Office Excel	<ul style="list-style-type: none">A spreadsheet used by financial and business professionals to retrieve data
• SQLCMD	<ul style="list-style-type: none">A command used by administrators for command line and batch files processing <pre>SQLCMD -S server\instance -i C:\script</pre>
• PowerShell	<ul style="list-style-type: none">An environment used by administrators for command line and batch processing

Key Points

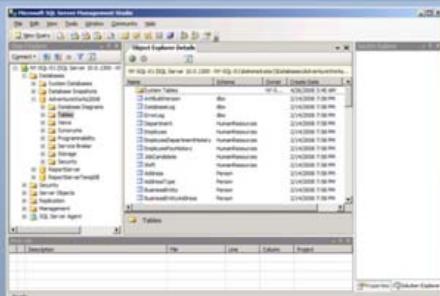
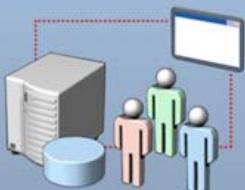
There are four main tools for querying SQL Server databases.

- The **bcp** utility bulk copies data between an instance of SQL Server and a data file in a user-specified format.
- The **sqlps** utility starts a Microsoft® PowerShell session with the SQL Server PowerShell provider and cmdlets loaded and registered.

Question: Have you used any of these tools for querying SQL Server databases?

An Introduction to SQL Server Management Studio

- **Support for writing and editing queries or scripts**
- **Integrated source control for solution and script projects**
- **Supports most administrative tasks for SQL Server**
- **An integrated Web browser for quick browsing**



Key Points

Microsoft® **SQL Server Management Studio** is a single, integrated environment for SQL Server Database Engine management and authoring. Within **SQL Server Management Studio**:

- The error and informational message box presents information, allows you to send Microsoft a comment about the messages, allows you to copy messages to the clipboard, and allows you to e-mail the messages to your support team.
- The SQL Server Management Studio has an activity monitor with filtering and automatic refresh.

What Is a SQL Server Solution?

SQL Server Management Studio provides two containers for managing database projects:



A solution includes projects and files that define the solution

A project is a set of files, plus related metadata

When you create a project, a solution is created to contain it

Key Points

A solution includes projects and files that define the solution.

- **SQL Server Management Studio** provides two containers for managing database projects such as scripts, queries, data connections, and files: solutions and projects.
- A project is a set of files, plus related metadata such as connection information. The files in a project depend on which SQL Server component the project is for.

Creating SQL Server Solutions

Solutions contain scripts, queries, connection information and files that you need to create your database solution

Use these containers to:

- **Implement source control on queries and scripts**
- **Manage settings for your solution**
- **Handle the details of file management**
- **Add items useful to multiple projects in to one solution**
- **Work on miscellaneous files independent from solutions**

Key Points

Solutions contain scripts, queries, connection information and files that you need to create your database solution.

- When you create a project, **SQL Server Management Studio** creates a solution to contain it.
- You can only open one solution at a time.
- **SQL Server Management Studio** does not allow you to create folders within projects.
- To organize your work, create multiple projects.
- You can use **Solution Explorer** to handle the details of file management while you focus on the items that make up your database solution.

Executing Queries in SQL Server Management Studio

- Executing queries occurs when in a query session by:
 - Selecting the Execute Icon
 - Pressing the F5 key
- Create queries by interactively entering them in a query window
- Load a file that contains T-SQL and then execute commands or modify then execute

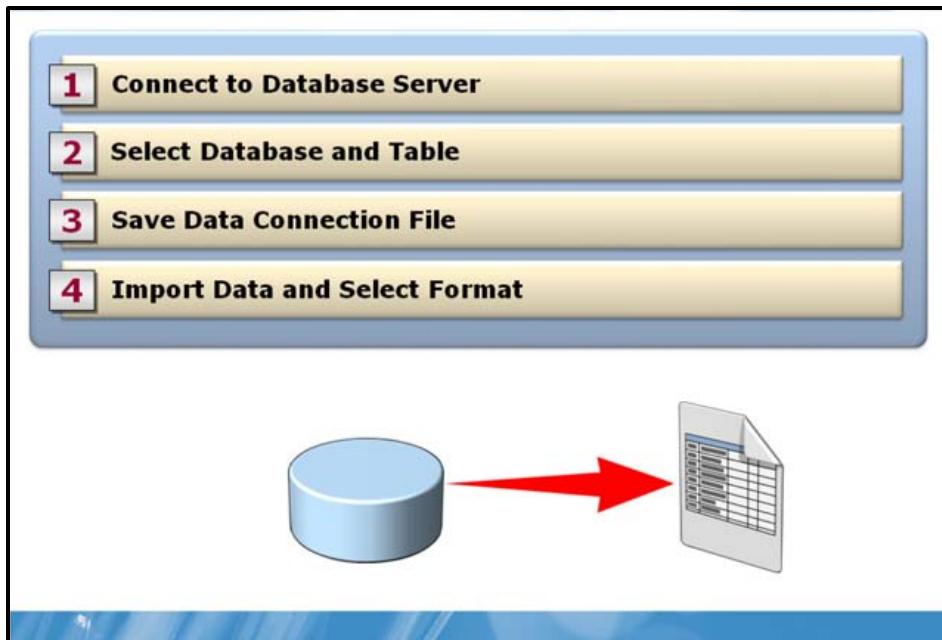


Key Points

In the Database Engine Query window, you can interactively code Transact-SQL and XQuery statements to query databases and change the data.

- The Database Engine Query Editor supports dynamic F1 help, auto-completion, code outlining, a T-SQL debugger, IntelliSense, and other productivity aids.
- In the **Object Explorer**, you can right-click tables or views and select menu items that let you select or edit rows.

Generating Reports in Microsoft Office Excel



Key Points

There are four steps to generating report in Microsoft Office Excel.

- You can use an Office Data Connection (.odc) file to connect to a SQL Server database from a Microsoft Office Excel 2007 file.
- MS Query can also be used to customize the type of information you would like to retrieve.

Lab: Using SQL Server Management Studio and SQLCMD

- Exercise 1: Explore the components and execute queries in SQL Server Management Studio
- Exercise 2: Start and use sqlcmd
- Exercise 3: Generate a report from a SQL Server database using Microsoft Office Excel

Logon information

Virtual machine	NY-SQL-01
User name	Student
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Exercise 1: Explore the Components and Execute Queries in SQL Server Management Studio

Scenario

You are the database administrator of Adventure Works. The Human Resources department requires that you generate several reports by using SQL Server 2008. You need to generate reports with a list of employee addresses sorted by departments or a list of addresses of employees residing in the United States.

The main tasks for this exercise are as follows:

1. Launch **SQL Server Management Studio**.
2. Navigate through the online help.
3. Resize, hide, and close **Object Explorer** and **Solution Explorer**.
4. Create a new solution and explore the solution objects in **Object Explorer**.
5. Add projects to an existing solution and create queries in the projects.
6. Connect to SQL Server and execute a query.
7. Use **Visual Query Builder** to return rows from a table.

► Task 1: Launch the SQL Server Management Studio

- Start **2778A-NY-SQL-01** and logon as **Student** using the password **Pa\$\$w0rd**.
- Start the **Server Management Studio** and connect to the server.

► Task 2: Navigate through online Help

- Navigate to **SQL Server Books Online** | **Getting Started** | **Initial Installation** | **Overview of SQL Server Installation** | **Features and Tools Overview** | **SQL Server Studios Overview** | **Introducing SQL Server**

Management Studio | Tutorial: SQL Server Management Studio | Lesson 1: Basic Navigation in SQL Server Management Studio.

- View the Lesson 1: Basic Navigation in SQL Server Management Studio page.
- View the Connecting with Registered Servers and Object Explorer page.

► **Task 3: Resize, hide, and close Object Explorer and Solution Explorer**

- Resize the Contents pane.
- Close the Help window.
- Auto Hide the **Solution Explorer** pane.
- Auto Hide the **Object Explorer** pane.

► **Task 4: Create a new solution and explore the solution objects in Object Explorer**

- Dock the **Object Explorer** pane.
- View the **AdventureWorks2008** tables.
- Create a new query.
- Dock the **Solution Explorer** pane.
- Add a new project.
- Name: **PersonAddress**
- Create a new connection.

► **Task 5: Add projects to an existing solution and create queries in the projects**

- Create a new query.
- Enter the query window.
 - Query: **USE AdventureWorks2008**
- Execute the query.
- Notice the active database is changed from master to AdventureWorks2008.
- Add additional text to the query window.
 - Query: **SELECT DISTINCT CITY FROM Person.Address**
- Execute the query.
- Rename **SQLQuery1.sql**.
 - Query: **Address.sql**
- Save **Address.sql**.
- Save **PersonAddress.ssmssqlproj**.
- Add a new project.
 - Name: **HumanResourcesDepartment**
- Create a new connection.
- Create a new query.
- Rename the query.
 - Name: **Department.sql**

- Enter the query window.
 - Query: **USE AdventureWorks2008**
- Execute the query.
- Notice the active database is changed from master to AdventureWorks2008.
- Add additional text to the query window.
 - Query: **SELECT Name, GroupName FROM HumanResources.Department**
- Execute the query.
- Save **Department.sql**.
- Save **HumanResourcesDepartment.ssmssqlproj**.
- Save solution as:
 - Name: **AdventureWorks2008.ssmssl**.
- Close **SQL Server Management Studio**.

► **Task 6: Connect to SQL Server and execute a Query**

- Start **Server Management Studio** and connect to the server.
- Open the file.
 - Folder: **PersonAddress**,
 - File: **Address.sql**
- Connect to Database Engine.
- Execute the query.

► **Task 7: Use Visual Query Builder to return rows from a table**

- Create a new query.
 - Database: **AdventureWorks2008**
- Open the **Query Designer**.
 - On the **Query** menu, click **Design Query in Editor**.
- Examine the Add Table dialog box.
- Select all columns.
- Execute the query.
- Close **SQL Server Management Studio**.

Results: After this exercise, you should have explored the components and executed queries in the SQL Server Management Studio.

Exercise 2: Start and Use sqlcmd

Scenario

You need to perform the query previously created using the **sqlcmd** utility.

The main tasks for this exercise are as follows:

1. Start the **sqlcmd** utility and connect to a default instance of SQL Server.
 2. Run a T-SQL script file by using **sqlcmd**.
 3. Run a T-SQL script file and save the output to a text file.
 4. Review the output file.
- **Task 1: Start the sqlcmd utility and connect to a default instance of SQL Server**
- Open **Command Prompt**.
 - Type **sqlcmd**.
 - End the sqlcmd session.
- **Task 2: Run a Transact-SQL script file by using sqlcmd**
- In the Command Prompt window, type a sqlcmd command:
 - Command: **sqlcmd -S NY-SQL-01 -i E:\MOD01\Labfiles\Starter\Department.sql**
- **Task 3: Run a Transact-SQL script file and save the output to a text file**
- In the Command Prompt window, type a sqlcmd command:
 - Command: **sqlcmd -S NY-SQL-01 -i E:\MOD01\Labfiles\Starter\Department.sql -o E:\MOD01\Labfiles\Solution\DeptList.txt**
 - Close **Command Prompt**.
- **Task 4: Review the output file**
- Start Microsoft® Windows Explorer.
 - Open the output file:
 - Directory: **E:\MOD01\Labfiles\Solution**
 - File: **DeptList.txt**
 - Close **Notepad**.
 - Close **Windows Explorer**.

Results: After this exercise, you should have started and used **sqlcmd** to create reports.

Exercise 3: Generate a Report from a SQL Server Database using Microsoft Office Excel

Scenario

You need to analyze the details of the newly hired employees by using a database diagram in **SQL Server Management Studio**.

The main tasks for this exercise are as follows:

1. Launch **Excel**.
2. Create a new data connection in the workbook.
3. Select the data to import and its format.
4. View the report.

► Task 1: Launch Excel

- Launch **Excel**.
 - Location: **All Programs | Microsoft Office | Microsoft Office Excel 2007**

► Task 2: Create a new data connection in the workbook

- Get external data from SQL Server.
 - Server name: **NY-SQL-01**

► Task 3: Select the data to import and its format

- Select the database and table.
 - Database: **AdventureWorks2008**
 - Table Name: **Address**
- Save data connection file and finish.
 - Description: **AdventureWorks2008 Addresses**
 - Friendly Name: **Addresses**
- Examine Authentication Settings.
- Click **Finish**.

► Task 4: View the report

- Notice the options for the Import Data dialog box.
- Accept the default settings.
- Examine the results.
- Close **Microsoft Office Excel**.
- Turn off **2778A-NY-SQL-01** and delete changes.

Results: After this exercise, you should create a report from a SQL Server database using Microsoft Office Excel.

Module Review and Takeaways

- Review Questions
- Best Practices
- Tools

Review Questions

1. What examples of a third tier client server architecture have you seen at work?
2. Which server management tool do we use to create optimal sets of indexes and partitions?
3. How would you quickly create graphs and reports from SQL data?
4. What are the two main parts of the relational database server?

Best Practices related to a particular technology area in this module

Supplement or modify the following best practices for your own work situations:

- Formatting
 - Capitalize reserve words.
 - Code should be indented properly.
- Naming Objects
 - Objects should be placed in square brackets.
 - [Customers]
 - Retain the case of the table names as in the database.
 - [OrderDetails]
 - Include schema in object names.
 - Sales.Customers
- Use ANSI SQL
- Comment Code

Tools

Tool	Use for	Where to find it
Microsoft SQL Server Management Studio	<ul style="list-style-type: none">Managing SQL server databases and tables.	Start All Programs Microsoft SQL Server 2008
SQL Server Business Intelligence Development Studio	<ul style="list-style-type: none">Managing SQL server applications.	Start All Programs Microsoft SQL Server 2008

Module 2

Querying and Filtering Data

Contents:

Lesson 1: Using the SELECT Statement	2-3
Lesson 2: Filtering Data	2-7
Lesson 3: Working with NULL Values	2-16
Lesson 4: Formatting Result Sets	2-20
Lesson 5: Performance Considerations for Writing Queries	2-27
Lab: Querying and Filtering Data	2-31

Module Overview

- Using the SELECT Statement
- Filtering Data
- Working with NULL Values
- Formatting Result Sets
- Performance Considerations for Writing Queries

The SELECT statement is the most fundamental statement type in Transact-SQL. By using a SELECT statement, you can retrieve data from a database for viewing or printing.

In this module, you will learn the format of the SELECT statement as well as some of the finer points of querying different types of data.

You will also learn some basics for formatting the selected data for easier interpretation.

The SELECT statement is the primary statement used for ad hoc queries against a database. Ad hoc queries are special purpose queries that are typically used a limited number of times, and therefore discarded after the data has been retrieved. These queries are typically created by end users or developers who need a quick view into the database.

Lesson 1

Using the SELECT Statement

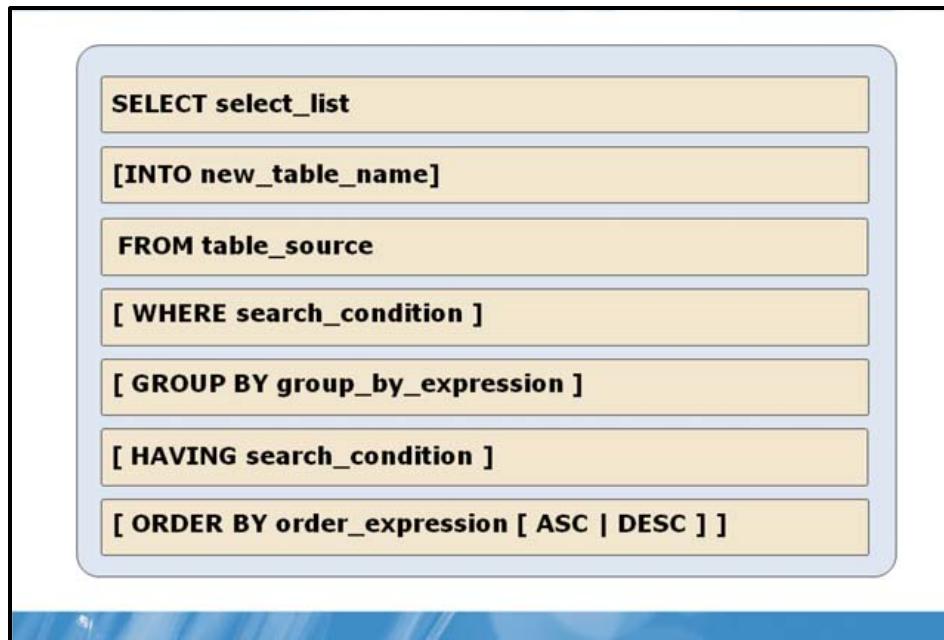
- Elements of the SELECT Statement
- Retrieving Columns in a Table

By using SELECT, you are able to retrieve data from a data source in its entirety or as a subset of the data.

This allows you to retrieve data relevant to your particular needs at a given time, and format the data for easy readability or print output.

In this lesson, you will learn how to use the basic SELECT and how to retrieve specific rows from the data source.

Elements of the SELECT Statement



Key Points

A SELECT statement requires:

- Select list of columns to retrieve
- FROM clause to establish the data source

Retrieving Columns in a Table

Displays All Columns in the Employee Table

```
USE AdventureWorks2008;
GO
SELECT *
FROM HumanResources.Employee
```

Displays Only FirstName, LastName and JobTitle Columns

```
USE AdventureWorks2008;
GO
SELECT FirstName, LastName, JobTitle
FROM HumanResources.Employee
```

Key Points

The select list of a SELECT statement can use a list of column names to retrieve or the '*' wildcard. If the '*' wildcard is used, then all columns in the data source will be retrieved and displayed.

The FROM clause allows you to designate the data source to retrieve the data rows from. A data source can be the name of table or a view in the database.

- Column names are designated in the select list.
- The '*' wildcard can be used to display all data columns.
- Retrieving all columns from a data source can degrade network performance.
- Retrieving all columns from a data source can make the result set difficult to read.
- The FROM clause contains the names of tables or views that the data is to be retrieved from.

Demonstration: Retrieve Data by Using the SELECT Statement

In this demonstration you will see how to:

- Use a simple query using the FROM clause to retrieve all data from a table.
- You will use a query that contains a select list and a FROM clause to retrieve all data in specific columns from a table.

Question: Considering the two ways of requesting columns from a table, which tables in the databases you work with are probably fine for using the generic * instead of listing each column?

Lesson 2

Filtering Data

- Retrieving Specific Rows in a Table
- Filtering Data by Using Comparison Operators
- Filtering Data by Using String Comparisons
- Filtering Data by Using Logical Operators
- Retrieving a Range of Values
- Retrieving a List of Values

Filtering data is the process of specifying criteria so that you only receive the specific data rows that you require. This can be done in Microsoft® SQL Server® by using comparison operators, string value comparisons, logical operators, ranges of values, and lists of values.

Nearly all queries and reports that you will be asked to create for your organization will require that the data be filtered to meet specific requirements. The requirements will differ by department and user.

Retrieving Specific Rows in a Table

Simple WHERE clause

```
USE AdventureWorks2008;
GO
SELECT BusinessEntityID AS 'Employee Identification Number',
HireDate, VacationHours, SickLeaveHours
FROM HumanResources.Employee
WHERE HireDate > '01/01/2000'
```

WHERE Clause Using a Predicate

```
USE AdventureWorks2008;
GO
SELECT FirstName, LastName, Phone
FROM Person.Person
WHERE EmailAddress IS NULL;
```



Key Points

The WHERE clause is used to set criteria for the result set. These criteria can be based on specific column data, ranges of data, or comparisons of the data in the data source. You can even test for NULL values, or empty columns.

SELECT statements do not require a WHERE clause. However, the result set can be very large if you do not filter the data by designating criteria.

- The WHERE clause is an optional part of the SELECT statement.
- WHERE designates criteria for the data that is to be retrieved.

Filtering Data by Using Comparison Operators

- Comparison operators test whether two expressions are the same.
- Comparison operators return a Boolean value of TRUE, FALSE, or UNKNOWN.

Scalar Comparison Operators

=, <>, >, >=, <, <=, !=

```
USE AdventureWorks2008;
GO
SELECT FirstName, LastName, MiddleName
FROM Person.Person
WHERE ModifiedDate >= '01/01/2004'
```

FirstName	LastName	MiddleName
Ken	Sánchez	J
Temi	Duffy	Lee
Roberto	Tamburello	NULL
Rob	Walters	NULL
Gail	Erickson	A

Key Points

Comparison operators check two values, or expressions, to see if they are the same. If the values are the same, the comparison returns a TRUE result. If they are different, a FALSE value is returned.

If either of the values is NULL, or empty, the comparison will return a value of UNKNOWN. This requires special handling that you will learn about later in this module.

- Comparison operators include:
 - = (equal)
 - <> or != (not equal)
 - > (greater than)
 - < (less than)
 - >= (greater than or equal)
 - <= (less than or equal)

Filtering Data by Using String Comparisons

- String Comparisons are used for data types of text, ntext, char, nchar, varchar, and nvarchar

- Predicates are available for full or partial match comparisons

`WHERE LastName = 'Johnson'`

`WHERE LastName LIKE 'Johns%n'`

`WHERE CONTAINS(LastName, 'Johnson')`

`WHERE FREETEXT(Description, 'Johnson')`

FirstName	LastName	MiddleName
Abigail	Johnson	NULL
Alexander	Johnson	M
Alexandra	Johnson	J
Alexis	Johnson	J
Alyssa	Johnson	K
Andrew	Johnson	F
Anna	Johnson	NULL

FirstName	LastName	MiddleName
Meredith	Johnsen	NULL
Rebekah	Johnsen	J
Ross	Johnsen	NULL
Willie	Johnsen	NULL
Abigail	Johnson	NULL
Alexander	Johnson	M
Alexandra	Johnson	J



Key Points

String comparisons are different than regular comparison operators. String comparisons allow you to check the entire string in a column or only part of the string by using wildcards. You also have special comparisons available that look to see if a given string is a substring of a value.

- = checks to see if the string value is the same as the expression.
- LIKE with wildcards checks for a specified pattern in a given string value.
 - % wildcard replaces any string of zero or more characters
 - _ (underline character) wildcard replaces any single character
- FREETEXT searches columns for values that match the meaning and not just the exact wording in the search condition.
- CONTAINS conducts a "fuzzy" search.

Question: When searching string data within a table which comparison do you think would be used the most? Why would this be most common?

Question: In the LIKE statement, where is the appropriate location to place the % sign and why?

Filtering Data by Using Logical Operators

- Logical operators are used to combine conditions in a statement**

Returns only rows with first name of 'John' and last name of 'Smith'

```
WHERE FirstName = 'John' AND LastName = 'Smith'
```

Returns all rows with first name of 'John' and all rows with last name of 'Smith'

```
WHERE FirstName = 'John' OR LastName = 'Smith'
```

Returns all rows with first name of 'John' and last name not equal to 'Smith'

```
WHERE FirstName = 'John' AND NOT LastName = 'Smith'
```



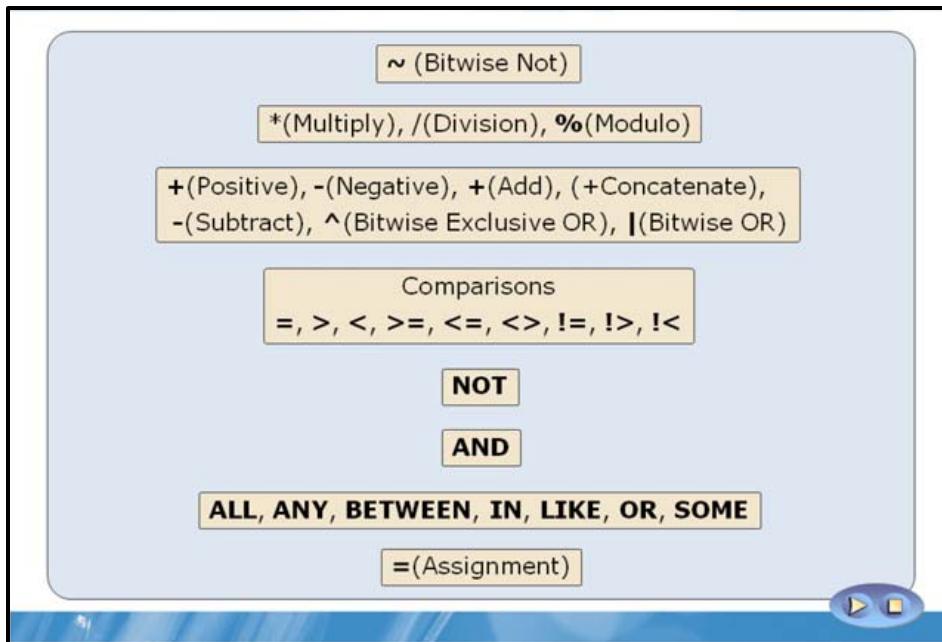
Key Points

Logical operators, AND, OR, and NOT allow expressions to be combined for filtering data.

- AND combines both expressions and requires them to be TRUE.
- OR tests to see if either one or both of the expressions returns TRUE.
- NOT tests to see if the first expression is TRUE and that the second expression is FALSE for each row returned in the result set.

Question: In a very large table why would you think the NOT operator might be the least efficient one to use?

Operator Precedence



Key Points

Operator precedence designates the order in which each operator is evaluated in an equation. The more complex the equation, the more important it is to consider precedence. Along with evaluation of operators based on the order above also consider:

- All operators that are at the same precedence level are evaluated from left to right.
- Parentheses are used to change the order of evaluation. Everything in parentheses is evaluated before the default precedence order is used.

Retrieving a Range of Values

- **BETWEEN** tests for data values within a range of values.

```
SELECT OrderDate, AccountNumber, SubTotal, TaxAmt  
FROM Sales.SalesOrderHeader  
WHERE OrderDate BETWEEN '08/01/2001' AND  
'08/31/2001'
```

- **BETWEEN** uses the same logic as \geq AND \leq

```
SELECT OrderDate, AccountNumber, SubTotal, TaxAmt  
FROM Sales.SalesOrderHeader  
WHERE OrderDate >= '08/01/2001'  
AND OrderDate <= '08/31/2001'
```

OrderDate	AccountNumber	SubTotal	TaxAmt
2001-08-01 00:00:00.000	10-4020-000018	39677.4848	3174.1988
2001-08-01 00:00:00.000	10-4020-000353	24299.928	1943.9942
2001-08-01 00:00:00.000	10-4020-000206	10295.8366	823.6669
2001-08-01 00:00:00.000	10-4020-000318	1133.2967	90.6637
2001-08-01 00:00:00.000	10-4020-000210	1086.6152	86.9292
2001-08-01 00:00:00.000	10-4020-000164	21923.9352	1753.9148
2001-08-01 00:00:00.000	10-4020-000697	24624.706	1969.9765
2001-08-01 00:00:00.000	10-4020-000191	12296.7218	982.9377

Key Points

Sometimes a range of values can be defined for filtering data. Instead of writing a WHERE clause with multiple expressions connected by the OR operator or using = and \leq expressions, you can use the BETWEEN predicate to specify a low and high value. This is equivalent to using a \geq AND \leq expressions.

BETWEEN is most commonly used to define data ranges, but is also used to define other ranges of values as well.

- BETWEEN is equivalent to \geq AND \leq .
- BETWEEN uses a low value and a high value to create the filter.
- BETWEEN includes the low and high values in the result set.

Retrieving a List of Values

- **IN** tests a column's values against a list of possible values.

```
SELECT SalesOrderID, OrderQty, ProductID, UnitPrice  
FROM Sales.SalesOrderDetail  
WHERE ProductID IN (750, 753, 765, 770)
```

- **IN** uses the same logic as multiple comparisons with the OR predicate between them

```
SELECT SalesOrderID, OrderQty, ProductID, UnitPrice  
FROM Sales.SalesOrderDetail  
WHERE ProductID = 750 OR ProductID = 753  
      OR ProductID = 765 OR ProductID = 770
```

SalesOrderID	OrderQty	ProductID	UnitPrice
43662	5	770	419.4589
43662	3	765	419.4589
43662	1	753	2146.962
43666	1	753	2146.962
43668	2	753	2146.962
43668	6	765	419.4589
43668	2	770	419.4589
43671	1	753	2146.962
43673	2	770	419.4589

Key Points

There are times when filtering for data that can have many possible values in a single column can get cumbersome. This will typically be expressions like the second example above that are all evaluated with an OR operator.

Transact-SQL has a special IN operator that allows the entry of possible values separated by commas. These values are then used like multiple OR expressions to filter the data. IN can also accept a single column result set from a sub query to populate the list.

- IN is the same as many expressions with OR between them.
- IN accepts a comma separated list of values.
- IN can accept a single column result set from a sub query.

Demonstration: Filter Data by Using Different Search Conditions

In this demonstration you will learn some of the ways that data can be filtered to show only the rows that meet specific criteria.

You will use:

- Comparison operators
- Range operators

Lesson 3

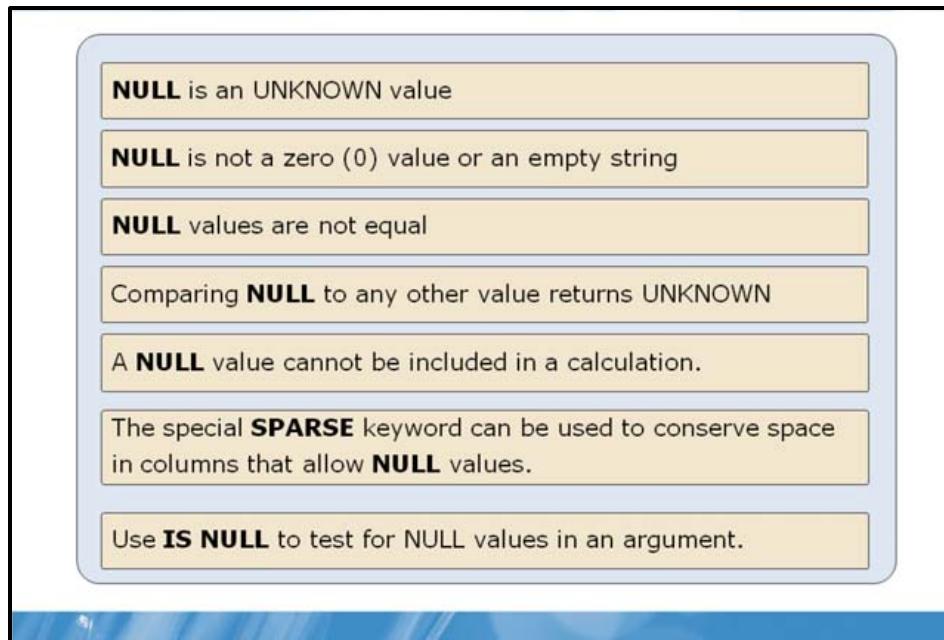
Working with NULL Values

- Considerations for Working with NULL Values
- Functions to Work with NULL Values

NULL values require special handling. NULL values are essentially undefined values they can create unexpected results when encountered in filtered columns. Usually, the result set will simply not include data that you wish to see just because of the presence of a NULL value.

By using IS NULL, IS NOT NULL, NULLIF, and COALESCE, you can test for and handle NULL values so that all of the data you are expecting will be included in the result set.

Considerations for Working with NULL Values



Key Points

NULL values can be the result of creating rows in a table and neglecting to set an explicit value for one of the columns. They can also occur in columns that are created with the SPARSE keyword to conserve space when an explicit value is not given.

Some characteristics of NULL values include:

- NULL returns a value of UNKNOWN.
- NULL is not a zero value.
- NULL is not an empty string.
- NULL compared to any other value or another NULL always returns UNKNOWN.
- NULL values in calculations will return an UNKNOWN value.
- IS NULL in the WHERE or HAVING clause tests for NULL values.

Functions to Work with NULL Values

ISNULL() returns a given value if the column value is NULL

```
SELECT Description, DiscountPct, MinQty, ISNULL(MaxQty, 0) AS  
    'Max Quantity'  
FROM Sales.SpecialOffer;
```

NULLIF() returns NULL if both specified expressions are equal

```
SELECT ProductID, MakeFlag, FinishedGoodsFlag,  
    NULLIF(MakeFlag,FinishedGoodsFlag) AS 'Null if Equal'  
FROM Production.Product  
WHERE ProductID < 10;
```

COALESCE() returns the first non NULL expression among its arguments, similar to a **CASE** statement

```
SELECT CAST(COALESCE(hourly_wage * 40 * 52, salary,  
    commission * num_sales) AS money) AS 'Total Salary'  
FROM wages
```



Key Points

Because NULL values can turn up unexpectedly, it is a good idea to have some tools to deal with them when they are encountered. Special functions such as ISNULL(), NULLIF(), and COALESCE() allow you to substitute actual values when NULL values are encountered.

- The ISNULL function assigns a value that will be returned if the argument value is NULL. The example above will return 0 if the MaxQty column value is NULL.
- The NULLIF function returns NULL if both arguments are equal. The example above will return the MakeFlag value if both values are different or a NULL value if both values are equal.
- The COALESCE function tests multiple arguments and returns the first one in the argument list that does not have a NULL value. The example above assumes that an employee can only have one wage type and returns the value of the wage assigned.

Question: When working with data, which data types would you expect to contain NULL values? How can you determine exactly which columns in your organization's tables are allowed to have NULL values?

Demonstration: Using Functions to Work with NULL Values

In this demonstration you will learn how to effectively deal with the unique issues created by NULL values by:

- Using the ISNULL() function to return an alternate value if the column value is NULL
- Using the NULLIF() function to return a NULL value if both columns being tested have the same value
- Using the COALESCE() function to test multiple columns or values and return the first one it determines to not have a NULL value

Question: Handling a NULL value is a very important part of retrieving accurate data for your organization. Considering these three functions, in what types of queries do you anticipate using each one?

Lesson 4

Formatting Result Sets

- Sorting Data
- Eliminating Duplicate Rows
- Labeling Columns in Result Sets
- Using String Literals
- Using Expressions

Retrieving the data is only the beginning. Once you have the data you are looking for, it needs to be formatted and grouped to make it more readable. Transact-SQL provides many ways to format your data including: grouping and sorting, eliminating duplicate rows and creating custom column labels.

Sorting Data

```
SELECT LastName, FirstName, MiddleName
FROM Person.Person
ORDER BY LastName, FirstName
```

LastName	FirstName	MiddleName
Abbas	Syed	E
Abel	Catherine	R
Abercrombie	Kim	NULL
Abercrombie	Kim	B
Abercrombie	Kim	NULL
Abolous	Hazem	E
Abolous	Sam	NULL
Acevedo	Humberto	NULL
Achong	Gustavo	NULL
Ackerman	Pilar	NULL
Ackerman	Pilar	G
Adams	Aaron	B
Adams	Adam	NULL
Adams	Alex	C
Adams	Alexandra	J
Adams	Allison	L
Adams	Amanda	P
Adams	Amber	NULL

Key Points

- Sorting of data is performed using the ORDER BY clause.
- ORDER BY takes a list of column names.
- Each column name is a field the data will be sorted on.
- If more than one column is present, the sorts will be nested.
- Ascending is the assumed sort order.
- Sort order is designated with ASC (ascending) and DESC (descending) keywords.

Eliminating Duplicate Rows

```
SELECT DISTINCT LastName, FirstName, MiddleName
FROM Person.Person
ORDER BY LastName, FirstName
```

LastName	FirstName	MiddleName
Abbas	Syed	E
Abel	Catherine	R.
Abercrombie	Kim	NULL
Abercrombie	Kim	B
Abokrouis	Hazem	E
Abokrouis	Sam	NULL
Acevedo	Humberto	NULL
Achong	Gustavo	NULL
Ackerman	Pilar	NULL
Ackerman	Pilar	G
Adams	Aaron	B
Adams	Adam	NULL
Adams	Alex	C
Adams	Alexandria	J
Adams	Allison	L
Adams	Amanda	P

Key Points

The data being queried will often contain duplicate rows. This is especially the case when you are querying for only a few of many columns in a table.

Duplicate data can make the result set very long, making it difficult to interpret properly.

To eliminate the duplicate rows in the result set, use the DISTINCT keyword in the SELECT list. When DISTINCT is used, the result set will contain only one instance of each unique row.

Question: Knowing that DISTINCT processes only those columns returned in the result set, are there situations where these results may not reflect an accurate representation of unique data?

Labeling Columns in Result Sets

- Aliases are used to create custom column headers in the result set display.
- You can rename actual or derived columns
- The optional **AS** clause can be added to make the statement more readable.
- Both statements below are equivalent

```
SELECT e.BusinessEntityID  
      AS 'Employee Identification Number'  
   FROM HumanResources.Employee AS e
```

```
SELECT e.BusinessEntityID  
      'Employee Identification Number'  
   FROM HumanResources.Employee e;
```

Employee Identification Number
109
4
9
11
158
263
267
270
2
46
106
119
203
269
271
272

Key Points

SELECT statements can be made easier to read by using aliases. Aliases can be assigned to columns in the select list and to data sources in the FROM clause.

Aliases for both columns and tables are within the SELECT statement as references to make typing and reading the statement simpler.

Not only can aliases make the statement easier to read, they can make the result set easier to read and interpret.

- Use column aliases in the result set as column headers. The example above uses column aliasing to change BusinessEntityID to the more meaningful "Employee".
- Use the optional AS clause between the column name and alias for easier readability.

Using String Literals

String Literals:

- Are constant values.
- Can be inserted into derived columns to format data.
- Can be used as alternate values in functions, such as the ISNULL() function.

```
SELECT (LastName + ' ' + FirstName + ' ' +
ISNULL(SUBSTRING(MiddleName, 1, 1), ' ')) AS Name
FROM Person.Person
ORDER BY LastName, FirstName, MiddleName
```

Name
Abbas, Syed, E
Abel, Catherine, R
Abercrombie, Kim, B
Abdeloua, Hazem, E
Ackerman, Pilar, G
Adams, Aaron, B
Adams, Alex, C
Adams, Alexandra, J
Adams, Allison, L
Adams, Amanda, P

Key Points

String literals are static strings that are inserted into derived columns, such as a comma between concatenated name parts. They can also be used as alternate values instead of column values in functions such as NULLIF() and COALESCE().

- Column and table aliases are not string literals.
- String literals can be used by functions as alternate return values.

Using Expressions

- Using mathematical expressions in SELECT and WHERE clauses
- Using functions in expressions

```
SELECT Name, ProductNumber, ListPrice AS OldPrice,  
      (ListPrice * 1.1) AS NewPrice  
  FROM Production.Product  
 WHERE ListPrice > 0 AND (ListPrice/StandardCost) > .8
```

```
SELECT Name, ProductNumber, ListPrice AS OldPrice,  
      (ListPrice * 1.1) AS NewPrice  
  FROM Production.Product  
 WHERE SellEndDate < GETDATE()
```

Key Points

Expressions can make a query more useful by allowing you to use calculations and functions to create derived columns and complex search arguments.

- Functions can be included in either the select list or the WHERE clause.
- Derived columns created by expressions require that you assign an alias.
- Calculations can be used to create complex mathematical criteria for filtering data.

Question: The above examples show how expressions can be used. What other ways do you think you will use expressions to create queries?

Demonstration: Format Result Sets

In this demo you will learn how to format result sets to:

- Assign custom column headings in the result set
- Concatenate string column values and use punctuation and spaces in the resulting value
- Create derived columns that return a calculated value
- Use expressions such as calculations and functions in the WHERE clause to create filtered result sets

Question: Formatting the result sets helps the reader interpret the data. What other reasons are there for creating result sets with a specific formatting?

Lesson 5

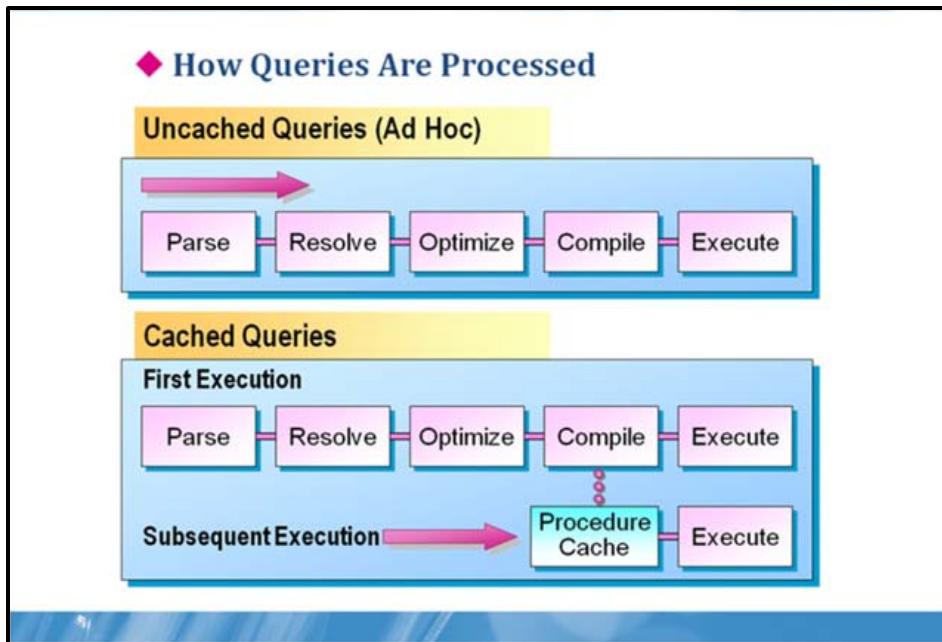
Performance Considerations for Writing Queries

- How SQL Server Processes Transact-SQL Queries
- Tips on Creating More Efficient Queries
- Solutions for Query Optimization

Queries are examined by the query optimizer before they are used to access data. The query optimizer parses the query and creates a list of steps that need to be taken to retrieve the data for the result set. This list of steps is then run through an algorithm that determines the best order of steps and method for each step.

There are a few simple things that can be done to help the query optimizer choose the best plan for executing a query. You will learn some of these techniques in this lesson.

How SQL Server Processes Transact-SQL Queries



Key Points

When the query optimizer examines a query it first parses the query into its logical units and then builds a list of steps that are needed to access the data source. The steps are then evaluated to determine a number of possible ways that they can be accomplished. Each of these possibilities is called an execution plane.

At this point, the optimizer calculates the resource cost of each execution plan and decides which one is least costly.

That plan is then passed on to the relational engine for execution and the result set is formatted and returned for viewing.

- The query optimizer may create many different execution plans for a query.
- Each plan is weighted as to its cost in resources and the lowest perceived cost plan is executed.
- You can include commands in the query to enable you to view the selected plan.

Tips on Creating More Efficient Queries

Some things to avoid:

- Leading wildcards - **LIKE '%an'**
- Negative condition tests - **...!=, NOT IN('California')**
- Expressions that use two columns in the same table
- Expressions that use column names in search arguments - **WHERE (Price + \$10) > \$20**

Some things to think about:

- Wildcards that are placed at the end or in the middle of an expression cost less in resources
- Use positive condition tests whenever possible
- Use calculations in search arguments sparingly
- Use indexed columns in search arguments whenever possible

Key Points

It is hard to know if your query is truly efficient unless you have a complex query with a considerable amount of data to go through. However, there are some things that you can avoid, or do that can help the optimizer find the best execution plan for any query.

- Things to avoid:
 - Columns in the WHERE clause that do not have an index.
 - Wildcards at the beginning of a LIKE clause.
 - Expressions that include column values in the WHERE clause.
- Things that will help:
 - Use constants for comparisons against column values whenever possible.
 - Use indexed columns in search arguments.
 - Use variables that have known values instead of variables that require frequent recalculation.

Solutions for Query Optimization

Solutions that can help with optimization:

- When using SQL Server built-in functions use constant values in the arguments whenever possible
- Build indexes on columns that will be used in search arguments
- Consider adding more memory to the server
- Use multiple processors on the server
- Avoid using multiple aliases for a single table in the query

Key Points

Sometimes the query is as efficient as you can make them but performance is not optimal. In these cases, you may have to take some of the following suggestions to your Database Administrator for help:

- Have the administrator look at the columns you use most often for filtering data and create or update indexes on those columns.
- Have the administrator create views for the most frequently used queries.

Lab: Querying and Filtering Data

- Retrieve data by using the SELECT statement
- Filter data by using different search conditions
- Use functions to work with NULL values
- Format result sets

Logon information

Virtual machine	NY-SQL-01
User name	Administrator
Password	Pa\$\$w0rd

Estimated time: 90 minutes

Exercise 1: Retrieve Data by Using the SELECT Statement

Scenario

Adventure Works management is attempting to learn more about their customers and they would like to contact each one with a survey. You have been asked to produce a list of contacts.

In this exercise, you will create simple SELECT statement queries.

The main tasks for this exercise are as follows:

1. Launch SQL Server Management Studio.
2. Generate a report by using the SELECT * statement with the FROM clause.
3. Generate a report by using the SELECT statement with the FROM clause.

► **Task 1: Launch SQL Server Management Studio**

- In the Lab Launcher, next to 2778A-NY-SQL-01, click **Launch**.
- Log on as **Administrator** with the password of **Pa\$\$w0rd**.

► **Task 2: Generate a report by using the SELECT * statement with the FROM clause**

- Start SQL Server Management Studio and connect to NY-SQL-01.
- In a new query window, create a query that uses the AdventureWorks2008 database and displays all columns and all rows from the Person.Person table.
- Execute the query and browse the results.

► **Task 3: Generate a report by using the SELECT statement with the FROM clause**

- In a new query window, type and execute an SQL statement that:
 - Accesses the Person.Person table in the AdventureWorks2008 database.

- Includes the FirstName, LastName, and MiddleName columns.
- Browse the result set in the Results pane and notice that only the specified columns appear.

Results: After this exercise, you should have learned how to create a basic SELECT statement to retrieve data from a table.

Exercise 2: Filter Data by Using Different Search Conditions

Scenario

The marketing department is conducting an audit of catalog inventory of socks and tights for a mail order campaign. In order to provide the best chance of creating orders, they have determined that most are willing to spend between \$50 and \$180 for these items. They have also found that most tights that are purchased in the store are size M and L. They have requested a list of all items the company carries within these parameters.

In this exercise, you will learn the different methods used to query for specific rows of data from a table.

The main tasks for this exercise are as follows:

1. Generate a report by using the SELECT statement with a COMPARISON operator.
2. Generate a report by using the SELECT statement with the AND and LIKE operators.
3. Generate a report by using the SELECT statement with the OR operator.
4. Generate a report by using the SELECT statement with the BETWEEN operator.
5. Generate a report by using the SELECT statement with the IN operator.

► **Task 1: Generate a report by using the SELECT statement with a COMPARISON operator**

- In a new query window enter and execute a SELECT statement that:
 - Accesses the Production.Product table in the AdventureWorks2008 database.
 - Includes the ProductNumber, Name, ListPrice, Color, Size, and Weight columns.
 - The ListPrice must be lower than \$100.
- Browse the result set.

► **Task 2: Generate a report by using the SELECT statement with the AND and LIKE operators**

- Change the SELECT statement so that it:
 - Accesses information in the Production.Product table of the AdventureWorks2008 database.
 - Includes ProductNumber, Name, ListPrice, Color, Size, and Weight
 - The ListPrice is less than \$100
 - The ProductNumber column begins with the string SO.
- Browse the result set and notice that all of the rows returned contain the word "SO".

► **Task 3: Generate a report by using the SELECT statement with the OR operator**

- Change the query to include rows with the TG as well as SO in the Name column.
- Execute the query.
- Browse the result set and note the additional rows for tights.

► **Task 4: Generate a report by using the SELECT statement with the BETWEEN operator**

- Change the query to select rows that:
 - Display rows with SO included in the Product Number.
 - Or rows of tights that have a list price between \$50 and \$180.
- Execute the query.

- Browse the result set and note that the number of rows has changed and the column data matches the new search conditions.

► **Task 5: Generate a report by using the SELECT statement with the IN operator**

- Add a search condition that limits 'tights' to sizes 'M' and 'L'.
- Execute the query.
- Note the additional filtering of the data rows within the result set.

Results: After this exercise, you should have learned how to use several different comparison operators to create reports specific to different user needs.

Exercise 3: Use Functions to Work with Null Values

Scenario

Upon checking through all of the catalog inventory records in the database, management has discovered that many items do not have a product line assigned to them. They would like a listing of all these items so they can make corrections in the system.

In this exercise, you will learn ways to locate and handle NULL values when querying data.

The main tasks for this exercise are as follows:

1. Generate a report by using the SELECT statement with the NULL function.
2. Generate a report by using the SELECT statement with the IS NULL function.
3. Generate a report by using the SELECT statement with the ISNULL function to rename values.
4. Generate a report by using the SELECT statement with the ISNULL function and a column alias.
5. Generate a report by using the SELECT statement with the ISNULL function and the COALESCE and CONVERT functions.

► **Task 1: Generate a report by using the SELECT statement with the NULL function**

- In a new query window, enter and execute a SELECT statement that:
 - Retrieves rows from the Production.Product table in the AdventureWorks2008 database.
 - Includes ProductNumber, Name, and Weight.
 - Uses the = operator to check ProductLine for a NULL value.
- Browse the result set and note the total number of rows.

► **Task 2: Generate a report by using the SELECT statement with the IS NULL function**

- Change the statement to use IS NULL to select rows with a value of NULL in the ProductLine column.
- Execute the query.
- Browse the result set and notice the number of rows returned and the data displayed in the ProductLine column.

► **Task 3: Generate a report by using the SELECT statement with the ISNULL function to rename values**

- In a new query window, enter and execute a SELECT statement that:
 - Accesses the Production.Product table in the AdventureWorks2008 database.
 - Displays the ProductNumber, Name, Weight, and ProductLine columns.
 - Use the ISNULL() function to display 'NA' when a NULL value is encountered in the ProductLine column.
- Browse the result set and note the additional column and values.
- Also make note of the column headings.

► **Task 4: Generate a report by using the SELECT statement with the ISNULL function and a column alias**

- Change the statement to add "Product Line" as a column heading for the ProductLine column.
- Execute the query.

- Note that the result set is identical but with a column heading over the new column.
- **Task 5: Generate a report by using the SELECT statement with the ISNULL function and the COALESCE and CONVERT functions**
- Rewrite the statement to use the COALESCE() function to create a new column named Measurement so that:
 - If the Weight column has a value it is shown in the Measurement column.
 - If the Weight column is NULL but the Size column is not NULL, display the value in the Measurement column.
 - If both columns have NULL values display 'NA'.
 - Execute the query.
 - Browse the result set and notice the new values in the 'Measurement' column.

Results: After this exercise, you should have learned to handle NULL values in a result set by identifying them and replacing them with alternate values when necessary.

Exercise 4: Formatting Result Sets

Scenario

- The marketing department needs a listing of items in the product file grouped by product line, description, and price.
- Management likes your original contacts list and would like you to refine it just a little so that the names are presented in a single column with the format of LastName, FirstName MiddleInitial.

In this exercise, you will learn how to format result sets to make them easier to read and interpret.

The main tasks for this exercise are as follows:

1. Format a result set by using the ORDER BY clause.
2. Format a result set by using the ORDER BY clause and the DESC keyword.
3. Format a result set by using the ORDER BY clause and the DISTINCT keyword.
4. Format a result set by concatenating strings.
5. Format a result set by concatenating strings and using column aliasing.
6. Format a result set by using the SUBSTRING function.

► Task 1: Format a result set by using the ORDER BY clause

- In a new query window, write a SELECT statement that:
 - Accesses the Production.Product table in the AdventureWorks2008 database.
 - Includes ProductNumber, Name, and Class.
 - Sorts the rows by the Class column.
 - Execute the query and browse the result set.

► Task 2: Format a result set by using the ORDER BY clause and the DESC keyword

- Rewrite the SELECT statement to:
 - Add the ListPrice column.
 - Sort the data by Class.
 - Sort by ListPrice in descending order within each class.
- Execute the query.
- Browse the new result set.

► Task 3: Format a result set by using the DISTINCT keyword

- Write a new SELECT statement that will:
 - Access the Production.Product table in the AdventureWorks2008 database.
 - Includes the Color column.
 - Shows only one row for each color.
 - For all products that have 'helmet' as part of the Name.
- Execute the query.
- Browse the result set.

► **Task 4: Format a result set by concatenating strings**

- Create a new listing that will:
 - Access the Person.Person table in the AdventureWorks2008 database.
 - Create a derived column made up of the LastName and FirstName columns concatenated with a comma and a space between them.
- Execute the query.
- Browse the result set and notice the new column has no meaningful heading.

► **Task 5: Format a result set by concatenating strings and using column aliasing**

- Rewrite the query to assign the column name 'Contacts'.
- Execute the query.
- Note the new column name in the result set.

► **Task 6: Format a result set by using the SUBSTRING function**

- Rewrite the statement so that it will search for all rows with the LastName beginning with 'Mac'.
- Use the SUBSTRING() function.
- Execute the query.
- Browse the result set.
- Turn off the 2778A-NY-SQL-01 virtual machine and discard changes.

Results: After this exercise, you should have learned how to format the result sets to make them more readable.

Module Review and Takeaways

- Review Questions
- Real-world Issues and Scenarios

Review Questions

1. When creating a WHERE clause by entering search conditions that include column data, what do you need to consider to help the query optimizer create the lowest cost execution plan? Why or how do these make a difference?
2. NULL values are present in many databases, usually by design. What can you do to discover where NULL values may be stored? Why is it important to know where NULL values can be encountered?

Real-world Issues and Scenarios

1. The DB Administrator has added a new column to the Person.Person table and you are asked to include this column in all queries and reports. But when you use the column in an expression in a search condition, few if any rows are returned. What is a likely cause? What can you do to help solve the issue?
2. You are asked to quickly create a report on a table with a large number of data rows in it. When writing the query, what are some things you can do to help the system retrieve the required rows as quickly as possible?

Module 3

Grouping and Summarizing Data

Contents:

Lesson 1: Summarizing Data by Using Aggregate Functions	3-3
Lesson 2: Summarizing Grouped Data	3-9
Lesson 3: Ranking Grouped Data	3-19
Lesson 4: Creating Crosstab Queries	3-26
Lab: Grouping and Summarizing Data	3-31

Module Overview

- Summarizing Data by Using Aggregate Functions
- Summarizing Grouped Data
- Ranking Grouped Data
- Creating Crosstab Queries

The ability to perform grouping and summaries of data is very important for a developer to deliver meaningful reports to users. This ability is delivered via aggregate functions, usually with grouped data. Aggregate functions are functions that operate on sets, or rows, of data, as opposed to set functions which operate on individual values. Aggregates can be applied in several ways, including the use of custom aggregates.

This module begins with an introduction of how to summarize data using aggregate functions. It then discusses how to apply these functions to grouped data. It also describes how to rank grouped data as well as using special operators to create crosstab queries.

Lesson 1:

Summarizing Data by Using Aggregate Functions

- Aggregate Functions Native to SQL Server
- Using Aggregate Functions with NULL Values
- CLR Integration, Assemblies
- Implementing Custom Aggregate Functions

Microsoft® SQL Server® 2008 provides several aggregate functions to help the developer summarize data. These functions can compute averages, sum values and count results. NULL values often require special attention when using aggregate functions. The ability to create custom aggregate functions is also present.

In this lesson, you will learn how to summarize data by using aggregate functions, whether built in or custom, and how to work with NULL values when using these functions.

Aggregate Functions Native to SQL Server

- Can be used in:
 - The select list of a SELECT statement
 - A COMPUTE or COMPUTE BY clause
 - A HAVING clause

```
USE AdventureWorks  
SELECT MAX(TaxRate)  
FROM Sales.SalesTaxRate  
GROUP BY TaxType;
```

Key Points

- Some common functions are AVG, MIN, MAX, SUM, COUNT, GROUPING and VAR.
- Aggregate functions perform calculations on a set, or group, of values and return .a single value.
- Note that the various aggregate functions work on differing date types – for instance AVG and SUM only work with numeric types.
- Aggregate functions usually obtain their sets of values to work on via the GROUP BY clause. When there is no GROUP BY, the functions get their groups from the entire table filtered by the WHERE clause.
- Aggregate functions can only appear in the SELECT statement, COMPUTE/COMPUTE BY clause and the HAVING clause.

Using Aggregate Functions with NULL Values

- Most aggregate functions ignore NULL values
- NULL values may produce unexpected or incorrect results
- Use the ISNULL function to correct this issue

```
USE AdventureWorks
```

```
SELECT AVG(ISNULL(Weight,0)) AS 'AvgWeight'  
FROM Production.Product
```

- The COUNT(*) function is an exception and returns the total number of records in a table

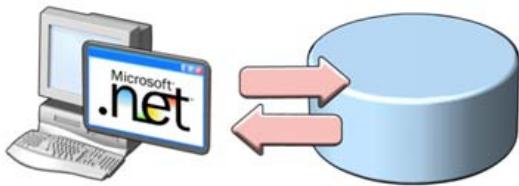
Key Points

- Most aggregate functions ignore NULL values. This can have unpredictable results. For instance, should you need to average records with potential NULL values, you may want to have the NULL values represent zero.
- The use of the ISNULL set function easily works .in cases where you want to include special handling for NULL values.
- Note the difference between COUNT(column) and COUNT(*). .If you want to count records that have a null value, you might get incorrect results because the COUNT() function, when used on a column, ignores null. In this case, COUNT(*) would be the correct choice.

Question: Why is it important to be aware of and handle NULL values in aggregate functions?

CLR Integration, Assemblies

- Introduced in SQL Server 2005
- Deploy custom developed CLR assemblies within the SQL Server Process
- Leverage the .NET 2.0 API



Key Points

- Beginning with Microsoft SQL Server 2005, SQL Server features the integration of the common language runtime (CLR) component of the .NET Framework for Microsoft® Windows®.
- Stored procedures, triggers, user-defined types, user-defined functions, user-defined aggregates, and streaming table-valued functions, can be written using any .NET Framework language (such as C# and VB.NET).
- Your custom code is compiled into CLR assemblies to later be loaded into the SQL Server Process via T-SQL commands.
- This allows you to leverage the entire .NET API in your triggers, stored procedures, or wherever CLR code is allowed.

Implementing Custom Aggregate Functions

- Custom programs in an assembly that are imported into SQL Server® using the integrated CLR
- Can be created using any .NET language such as C# and VB

```
CREATE ASSEMBLY StringUtilities FROM  
'PathToAssembly\StringUtilities.dll'
```

```
WITH PERMISSION_SET=SAFE;
```

```
CREATE AGGREGATE Concatenate(@input  
nvarchar(4000))
```

```
RETURNS nvarchar(4000)
```

```
EXTERNAL NAME [StringUtilities].[Concatenate]
```

Key Points

- Custom aggregate functions are developed for use in SQL Server via CLR Integration.
- Custom aggregates are essentially programs you create in .NET to be used to perform complex logic and calculations.
- You compile custom aggregates into .NET assemblies and load them into a CLR enabled SQL Server by using the CREATE ASSEMBLY statement.
- You then create the aggregate with the CREATE AGGREGATE statement.
- Once a User Defined Aggregate (UDA) is loaded into a SQL Server instance, it is readily used like any other built in aggregate function.

BENEFITS TO USING CLR INTEGRATION

Efficient alternative to extended stored procedures

Ability to leverage comprehensive .NET class library APIs.

Seamless debugging with other parts of SQL Server

Question: Why use a custom aggregation function if you're comfortable with handling the logic in T-SQL?

Demonstration: Using Common Aggregate Functions

- This demonstration shows how to use some of the common aggregate functions in SQL Server 2008. The Human Resources (HR) department of Adventure Works requires information about employees. This would be an excellent situation to show the usage of several aggregate functions.

Question: What would be the best aggregate function to use when counting records in a table?

Question: How can you restrict one of the sample queries in this demo to only account for a certain range of employees?

Lesson 2

Summarizing Grouped Data

- Using the GROUP BY clause
- Filtering Grouped Data by Using the HAVING Clause
- Building a Query for Summarizing Grouped Data – GROUP BY
- Examining How the ROLLUP and CUBE Operators Work
- Using the ROLLUP and CUBE Operators
- Using the COMPUTE and COMPUTE BY Clauses
- Building a Query for Summarizing Grouped Data - COMPUTE
- Using GROUPING SETS

Aggregate functions can be applied to whole and unfiltered tables, but these functions really show their usefulness when used in conjunction with grouped sets of data. In SQL Server, the grouping of data can come via the use of several T-SQL clauses and operators.

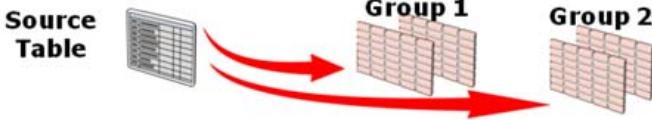
In this lesson, you will learn how to summarize grouped data by using these clauses and operators.

Using the GROUP BY Clause

- Specifies the groups into which the output rows must be placed
- Calculates a summary value for aggregate functions

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

SalesOrderID	SubTotal
1	23761
2	45791
3	75909
4	19900



Key Points

- The GROUP BY clause is used to group rows together into result sets.
- When performing grouping in a query, the query results will be grouped by the columns listed in the GROUP BY clause. It is this grouping that creates the sets that aggregate functions work with. Hence if you want to SUM the LineTotal for each SalesOrderDetail record, you would want to group on a column that identifies each sales order uniquely; the SalesOrderID column in this case.
- When using the GROUP BY clause, all columns in the SELECT list not part of an aggregate expression will be used to group the results via the GROUP BY clause. If you don't want to group on a column, don't put it in the SELECT list.
- NULL values are included in a column's grouping - all NULL values are considered equal and thus placed into their own group.

Question: Why might you want to group results... is this absolutely necessary?

Filtering Grouped Data By Using the HAVING Clause

- Specifies a search condition for a group
- Can be used only with the SELECT statement

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

SalesOrderID	SubTotal
43875	121761.939600
43884	115696.331324
44518	126198.336168
44528	108783.587200
44530	104958.806836
44795	104111.515642
46066	100378.907800
...	

- The HAVING clause is an optional clause that can be used to narrow down a result set by setting criteria on aggregate values, effectively replacing the WHERE clause.
- The HAVING clause is used to filter rows *after* the grouping has been applied, but *before* the results are returned to the client.
- You can only include aggregate expressions and columns that are listed in the GROUP BY clause.
- It is a good practice to filter your aggregate results using a HAVING clause, not in a WHERE clause.
- The HAVING clause can also be used like a WHERE clause if GROUP BY is not used in the query.

Question: Are HAVING and WHERE essentially the same?

Building a Query for Summarizing Grouped Data - GROUP BY

- **GROUP BY**

```
SELECT A.City, COUNT(E.BusinessEntityID) EmployeeCount
FROM HumanResources.Employee E
INNER JOIN Person.Address A ON E.BusinessEntityID =
A.AddressID
GROUP BY A.City ORDER BY A.City;
```

City	EmployeeCount
Bellevue	35
Berlin	1
Bordeaux	1
Bothell	22
Calgary	1
Cambridge	2
...	

- **GROUP BY with HAVING clause**

```
SELECT DATEPART(yyyy,OrderDate) AS 'Year' ,SUM(TotalDue) AS
'Total Order Amount'
FROM Sales.SalesOrderHeader
GROUP BY DATEPART(yyyy,OrderDate)
HAVING DATEPART(yyyy,OrderDate) >= '2003'
ORDER BY DATEPART(yyyy,OrderDate)
```

Year	Total Order Amount
2003	54307615.0868
2004	32196912.4165

Key Points

GROUP BY	HAVING
Provides summarized data sets for aggregation functions.	Can be used in lieu of WHERE.
When using aggregation functions, columns in SELECT list must either be in a function or in the GROUP BY.	Filters grouping data after the grouping has occurred but before sent as results.

Examining How the ROLLUP and CUBE Operators Work

- ROLLUP and CUBE generate summary information in a query

- ROLLUP generates a result set showing the aggregates for a hierarchy of values in selected columns

```
SELECT a, b, c, SUM ( <expression> )
FROM T
GROUP BY ROLLUP (a,b,c)
```

- CUBE generates a result set that shows the aggregates for all combination of values in selected columns

```
SELECT a, b, c, SUM (<expression>)
FROM T
GROUP BY CUBE (a,b,c)
```

Key Points

- Both ROLLUP and CUBE are new operators used in the GROUP BY clause. They both generate summary information in a query.

ROLLUP	CUBE
Generates subtotal, or super-aggregate, rows and grand total row	Generates ROLLUP super-aggregate rows and cross tab rows.
One row with a subtotal is generated for each unique combination of the columns in the select list.	Outputs a grouping for all combinations of expressions.
Column order affects output groupings (rollup direction)	Column order doesn't affect CUBE output.

Using the ROLLUP and CUBE Operators

```
SELECT ProductID, Shelf, SUM(Quantity) AS QtySum  
FROM Production.ProductInventory  
WHERE ProductID < 6  
GROUP BY ROLLUP(ProductID, Shelf)
```



ProductID	Shelf	QtySum
1	A	761
1	B	324
1	NULL	1085
2	A	791
2	B	318
2	NULL	1109
3	A	909
3	B	443
3	NULL	1352
4	A	900

```
SELECT ProductID, Shelf, SUM(Quantity) AS QtySum  
FROM Production.ProductInventory  
WHERE ProductID < 6  
GROUP BY CUBE(ProductID, Shelf)
```



ProductID	Shelf	QtySum
1	A	761
2	A	791
3	A	909
4	A	900
NULL	A	3361
1	B	324
2	B	318
3	B	443
4	B	442
NULL	B	1507

Key Points

- The ROLLUP example generates a subtotal for each unique combination of values for (a,b,c), (a,b), and (a). A grand total row is also created.
- The CUBE example generates a row for each unique combination of values of (a,b,c), (a,b), (a,c), (b,c), (a), (b), and (c) with a subtotal for each row and a grand total row.

Demonstration: How to Use the ROLLUP and CUBE Operators

- This demonstration shows how to use the ROLLUP and CUBE operator extensions of the GROUP BY SELECT clause. You have been asked to summarize some information in the AdventureWorks database. This would be an excellent usage of these new operators.

Question: What could be some determining factors on whether you chose to use a ROLLUP vs. a CUBE query?

Examining the COMPUTE and COMPUTE BY Clauses

- COMPUTE generates additional summary rows in a non-relational format
- COMPUTE BY generates control-breaks and subtotals in the result set
- Both COMPUTE and COMPUTE BY can be used in the same query



Key Points

- COMPUTE generates totals that appear as additional summary columns at the end of the result set.
- COMPUTE creates two result sets for each group. The first result set contains detail rows from the SELECT statement for that group, the second has one row that contains the subtotals of the aggregate functions specified in the COMPUTE clause.
- COMPUTE BY allows both detail (subtotal) and summary (grand total) rows to be created with one SELECT statement. Use of COMPUTE BY requires the ORDER BY clause.
- You should avoid the COMPUTE and COMPUTE BY clauses as they have been deprecated in favor of the ROLLUP operator.

Building a Query for Summarizing Group Data - COMPUTE

• COMPUTE

```
SELECT SalesOrderID, UnitPrice, UnitPriceDiscount  
FROM Sales.SalesOrderDetail  
ORDER BY SalesOrderID  
COMPUTE SUM(UnitPrice), SUM(UnitPriceDiscount)
```

SalesOrderID	UnitPrice	UnitPriceDiscount
1	50	200
2	135	350
3	NULL	1085

sum	Sum
185	1635

• COMPUTE BY

```
SELECT SalesPersonID, CustomerID, OrderDate, SubTotal, TotalDue  
FROM Sales.SalesOrderHeader  
ORDER BY SalesPersonID, OrderDate  
COMPUTE SUM(SubTotal), SUM(TotalDue) BY SalesPersonID
```

Key Points

- Both the COMPUTE and COMPUTE BY examples contain aggregation functions in the COMPUTE clause.
- The first example simply produces grand totals for the SalesOrderDetail table.
- In the second example, note the specification of group boundary of SalesPersonID as well as the ORDER BY clause.
- The second example returns subtotals demarcated by SalesPersonID via the BY statement. This is followed by the grand totals, as in the first example.

Question: What advantages does using the COMPUTE statement provide?

Using GROUPING SETS

- New GROUP BY operator that aggregates several groupings in one query
- Eliminates multiple GROUP BY queries

```

SELECT T.[Group] AS 'Region', T.CountryRegionCode AS 'Country'
      ,S.Name AS 'Store', H.SalesPersonID
      ,SUM(TotalDue) AS 'Total Sales'
  FROM Sales.Customer C
  INNER JOIN Sales.Store S
    ON C.StoreID = S.BusinessEntityID
  INNER JOIN Sales.SalesTerritory T
    ON C.TerritoryID = T.TerritoryID
  INNER JOIN Sales.SalesOrderHeader H
    ON C.CustomerID = H.CustomerID
 WHERE T.[Group] = 'Europe'
   AND T.CountryRegionCode IN('DE', 'FR')
   AND SUBSTRING(S.Name,1,4)IN('Vers', 'Spa')
 GROUP BY GROUPING SETS
      (T.[Group], T.CountryRegionCode, S.Name, H.SalesPersonID)
 ORDER BY T.[Group], T.CountryRegionCode, S.Name, H.SalesPersonID;

```

Region	Country	Store
NULL	NULL	NULL
NULL	NULL	NULL
NULL	NULL	NULL
NULL	NULL	Spa and Exercise Outfitters
NULL	NULL	Versatile Sporting Good Company
NULL	DE	NULL
NULL	FR	NULL
Europe	NULL	NULL

- Supports additional options as ability to use with ROLLUP and CUBE operators

Key Points

- GROUPING SETS is a new GROUP By operator that changes how one can perform relational grouping. Instead of multiple queries with GROUP BY clauses with a UNION ALL, GROUPING SETS lets you do the same thing in a single query.
- GROUPING SETS have additional support for optional grand total rows and can be used in conjunction with ROLLUP and CUBE operators.
- Since GROUPING SETS provide a new angle on grouping data, there are many equivalent operations to queries which use the standard GROUP BY clause.

Lesson 3

Ranking Grouped Data

- What Is Ranking?
- Ranking Data by Using RANK
- Ranking Data by Using DENSE_RANK
- Ranking Data by Using ROW_NUMBER
- Ranking Data by Using NTILE
- Categorizing the Ranking Functions Based On Their Functionality

Ranking refers to the numbering of rows in a set of data given a type of ranking and set of qualifiers. Ranking can further be applied to a user defined window, or partition, of data.

In this lesson, you will learn how to apply the ranking functions to sets of data in SQL Server.

What Is Ranking?

- Ranking computes a value for each row in user-specified set of rows.

- Ranking functions include:

- RANK

- DENSE_RANK

- NTILE

- ROW_NUMBER

- Can be used for:

- Arbitrary row numbering

- Product Popularity

- Best Customers

Key Points

- SQL Server 2008 provides four ranking functions. The four ranking functions each provide differing ranking output. For instance, some functions return consecutive numbers where others do not.
- Ranking functions operate similarly to aggregate functions when it comes to partitions, or "windows", of data.
- Ranking functions return a ranking value for each row in a partition depending on what type of ranking function is used based on its relationship to other rows and the values of the partitioned column.

Question: Does your organization currently use ranking in its applications?

Ranking Data by Using RANK

```
SELECT P.Name Product, P.ListPrice, PSC.Name Category,  
RANK() OVER(PARTITION BY PSC.Name  
ORDER BY P.ListPrice DESC)  
AS PriceRank  
FROM Production.Product P  
JOIN Production.ProductSubCategory PSC  
ON P.ProductSubCategoryID = PSC.ProductSubCategoryID
```

Product	ListPrice	Category	PriceRank
Men's Bib-Shorts, S	89.99	Bib-Shorts	1
Men's Bib-Shorts, M	89.99	Bib-Shorts	1
Men's Bib-Shorts, L	89.99	Bib-Shorts	1
Hitch Rack - 4-Bike	120.00	Bike Racks	1
All-Purpose Bike Stand	159.00	Bike Stands	1
Mountain Bottle Cage	9.99	Bottles and Cages	1
Road Bottle Cage	8.99	Bottles and Cages	2

Key Points

- In this example, a RANK query is used to assign rank values based on quantity values in the database. You could use this same approach to provide a top ten list of customers based on purchase amounts.
- Note how the ORDER BY in the OVER clause orders the RANK while the ORDER BY of the SELECT orders the result set.
- Tied rows receive the same rank.
- Returns the rank of each row within the specified partition of a result set.

Ranking Data by Using DENSE_RANK

```
SELECT P.Name Product, P.ListPrice, PSC.Name Category,
DENSE_RANK()
OVER(PARTITION BY PSC.Name ORDER BY P.ListPrice DESC)
AS PriceRank
FROM Production.Product P
JOIN Production.ProductSubCategory PSC
ON P.ProductSubCategoryID = PSC.ProductSubCategoryID
```

Product	ListPrice	Category	PriceRank
HL Mountain Handlebars	120.27	Handlebars	1
HL Road Handlebars	120.27	Handlebars	1
HL Touring Handlebars	91.57	Handlebars	2
ML Mountain Handlebars	61.92	Handlebars	3
HL Headset	124.73	Headsets	1
ML Headset	102.29	Headsets	2
LL Headset	34.20	Headsets	3

Key Points

- In this example, a DENSE_RANK query is used to provide categorized ranking without any gaps in the ranking. Exactly like RANK, two items can still tie in their ranking.
- DENSE_RANK works the same as RANK except there are no gaps in the ranking sequence.
- The rank of a row is one plus the number of distinct ranks that come before the row in question.

Ranking Data by Using ROW_NUMBER

```
SELECT ROW_NUMBER()
OVER(PARTITION BY PC.Name ORDER BY ListPrice)
AS Row, PC.Name Category, P.Name Product, P.ListPrice
FROM Production.Product P
JOIN Production.ProductSubCategory PSC
ON P.ProductSubCategoryID = PSC.ProductSubCategoryID
JOIN Production.ProductCategory PC
ON PSC.ProductCategoryID = PC.ProductCategoryID
```

Row	Category	Product	ListPrice
1	Accessories	Patch Kit/8 Patches	2.29
2	Accessories	Road Tire Tube	3.99
1	Bikes	Road-750 Black, 44	539.99
2	Bikes	Road-750 Black, 44	539.99

Key Points

- In this example, ROW_NUMBER is being used to categorize and rank rows in a result set. The sequence in which the rows are assigned row numbers is determined by the ORDER BY clause.
- Can be used with or without a PARTITION BY clause which, if used, divides the result set into partitions to which ROW_NUMBER is applied.

Ranking Data by Using NTILE

```
SELECT NTILE(3) OVER(PARTITION BY PC.Name ORDER BY ListPrice)
AS PriceBand, PC.Name Category, P.Name Product, P.ListPrice
FROM Production.Product P
JOIN Production.ProductSubCategory PSC
ON P.ProductSubCategoryID = PSC.ProductSubCategoryID
JOIN Production.ProductCategory PC
ON PSC.ProductCategoryID = PC.ProductCategoryID
```

PriceBand	Category	Product	ListPrice
1	AccessoriesPatch	Kit/8 Patches	2.29
1	AccessoriesRoad	Tire Tube	3.99
2	AccessoriesLL	Road Tire	21.49
2	AccessoriesML	Road Tire	24.99
3	AccessoriesSport-100	Helmet, Blue	34.99
3	AccessoriesHL	Mountain Tire	35.00
1	Bikes	Road-750 Black, 44	539.99
1	Bikes	Road-650 Red, 48	782.99
2	Bikes	Road-650 Red, 52	782.99

Key Points

- NTILE has the capability to distribute rows in an ordered partition into a specified number of groups.
- In this example, the determining factor that is used to decide the groupings is specified as the ListPrice—as that is the column the partition is ordered by. The groups are partitioned by Category name.
- NTILE is often used in data warehousing.
- Distributes rows in an ordered partition on a specified number of groups.
- For each row, NTILE returns the number of the group to which the row belongs.

Categorizing the Ranking Functions Based On Their Functionality

Feature	RANK	DENSE_RANK	NTILE	ROW_NUMBER
Rows equally distributed among groups			✓	
Rows of equal rank are assigned the same value	✓	✓		
Sequential numbering of rows within a partition				✓

Key Points

This table describes the different advantages the various rank functions have.

Some possible business scenarios for the various ranking functions are:

- NTILE
 - Data Warehousing
 - Divide sales teams into equal sized groupings based on sales figures
- RANK, DENSE_RANK
 - Rank best sales days of the year
 - Rank best selling products
- ROW_NUMBER
 - Number your contacts in a custom ordering

Question: What ranking function would be most useful in ranking the teams in a sports league where you don't want gaps in the rankings?

Lesson 4

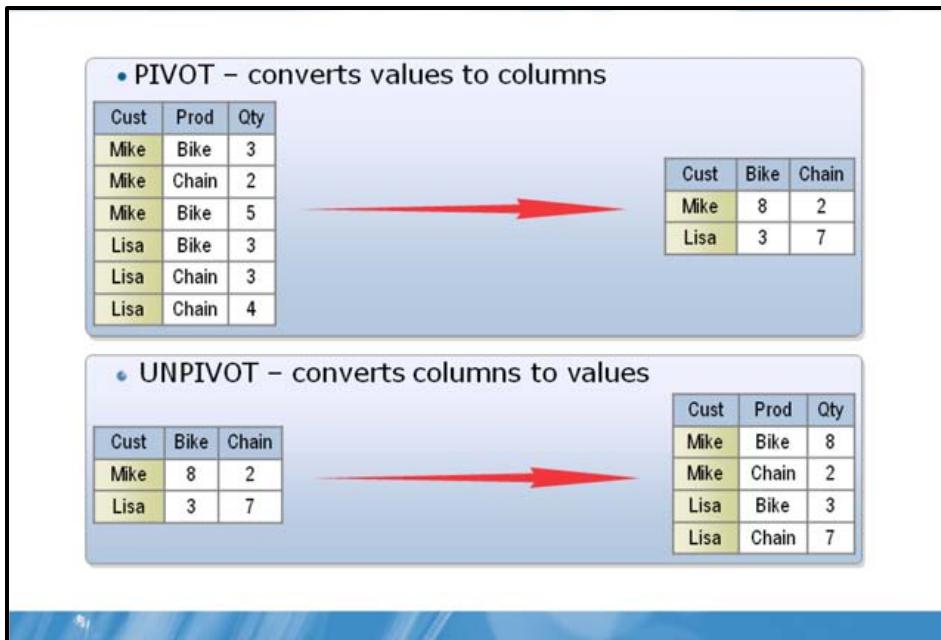
Creating Crosstab Queries

- How the PIVOT and UNPIVOT Operators Work
- Using the PIVOT Operator
- Using the UNPIVOT Operator
- Grouping and Summarization Features New to SQL Server 2008

The creation of crosstab queries can be a laborious and error prone process without help from the platform. SQL Server provides the PIVOT operator to ease this process and the UNPIVOT operator to reverse data that has previously been PIVOTed.

In this lesson, you will learn how to apply both the PIVOT and UNPIVOT operators to data in SQL Server.

How the PIVOT And UNPIVOT Operators Work



Key Points

- PIVOT rotates a table-valued expression by turning the unique values from one column in the expression into multiple columns in the output, and performs aggregations where they are required on any remaining column values that are wanted in the final output.
- PIVOT works by taking values such as EmployeeID and translating them into columns. Note that PIVOT is commonly used in generating cross tab reports.
- UNPIVOT performs the opposite operation to PIVOT by rotating columns of a table-valued expression into column values.

Using the PIVOT Operator

• PIVOT – converts values to columns

Cust	Prod	Qty
Mike	Bike	3
Mike	Chain	2
Mike	Bike	5
Lisa	Bike	3
Lisa	Chain	3
Lisa	Chain	4

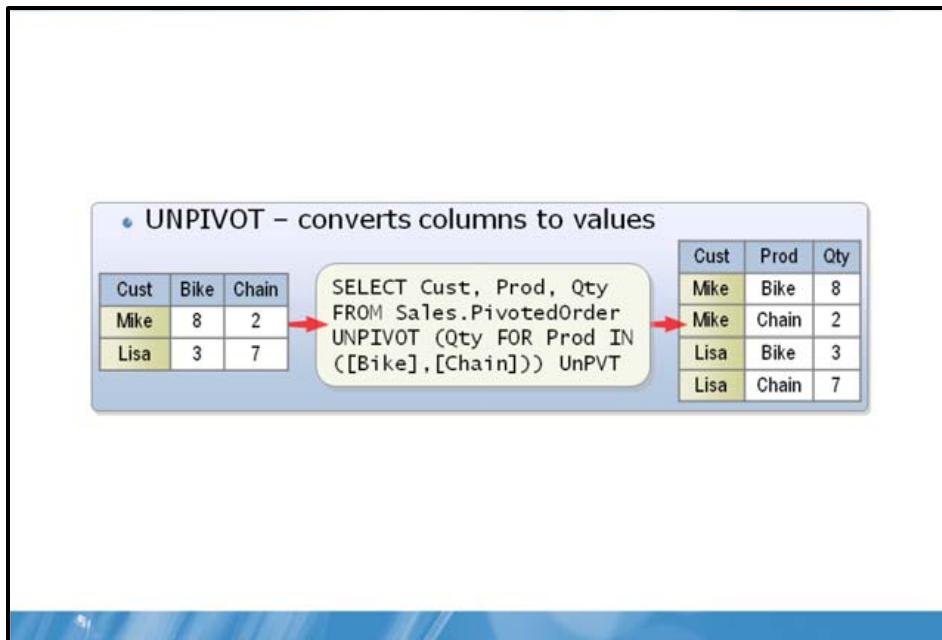
SELECT * FROM Sales.Order
PIVOT (SUM(Qty) FOR Prod
IN ([Bike],[Chain])) PVT

Cust	Bike	Chain
Mike	8	2
Lisa	3	7

Key Points

- You can see that this example is querying the PurchaseOrderHeader table to determine the number of purchase orders placed by certain employees, ordered by vendor.
- The syntax for PIVOT is simpler than would otherwise be used in a complex series of SELECT...CASE statements to achieve similar results.
- A common scenario where PIVOT can be useful is when you want to generate cross-tabulation reports to summarize data.

Using the UNPIVOT Operator



Key Points

- Here you see an example of the use of UNPIVOT to break crosstab result sets into table-valued result sets.
- UNPIVOT does not reproduce the original table-valued expression result because rows have been merged. If you page back to the PIVOT example, you will see that the results produced by the UNPIVOT operation do not match exactly the input that the PIVOT operation started with.

Grouping and Summarization Features New to SQL Server 2008

- GROUPING SETS Operator

- ROLLUP Operator

- CUBE Operator

- GROUPING_ID Function

Key Points

- Probably the most effective new feature for performing relational grouping is the GROUPING SETS operator addition to the GROUP BY clause. This new operator allows you to perform several grouping sets in one query. The older method was to perform several GROUP BY queries and then perform a UNION ALL between them.
- The ROLLUP and CUBE operators aren't new features per se but rather syntax changes. Prior to SQL Server 2008, you would specify WITH ROLLUP or WITH CUBE. In SQL Server 2008, you use ROLLUP and CUBE almost like functions as you pass the grouped columns into these operators.
- There is a new function, GROUPING_ID which returns the level of grouping for a particular column. This function is an enhancement from the older GROUPING() function which returns a 0 or a 1 to indicate if the column is aggregated or not.

Lab: Grouping and Summarizing Data

- Exercise 1: Summarizing Data by Using Aggregate Functions
- Exercise 2: Summarizing Grouped Data
- Exercise 3: Ranking Grouped Data
- Exercise 4: Creating Crosstab Queries

Logon information

Virtual machine	NY-SQL-01
User name	Administrator
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Exercise 1: Summarizing Data by Using Aggregate Functions

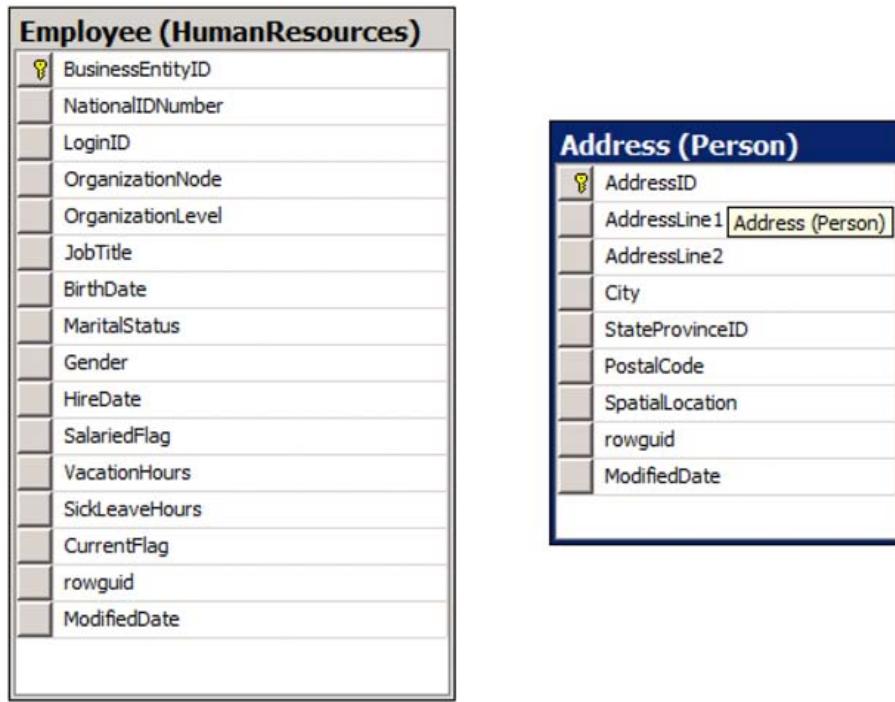
Scenario

You are the database administrator of Adventure Works. The HR department wants you to prepare a report on the number of vacation hours of the vice presidents of the company, and another report on the number of employees with incomplete addresses.

In this exercise, you will use built in aggregate functions to summarize data in different ways. You will also work around NULL values in your summaries.

This exercise's main tasks are:

1. Start the **2778A-NY-SQL-01** virtual machine, log on as **Administrator**, and launch **SQL Server Management Studio**.
2. Create and execute a query that displays a single summary value for all rows.
3. Create and execute a second query that displays a single summary value for all rows.
4. Create and execute a third query that computes a total.
5. Modify a query to eliminate NULL values.



- ▶ **Task 1: Start the 2778A-NY-SQL-01 virtual machine, log on as Administrator, and launch SQL Server Management Studio**
 - Start **6235A-NY-SQL-01**, and log on as **Administrator** with the password of **Pa\$\$w0rd**.
 - Launch **SQL Server Management Studio**.
- ▶ **Task 2: Create and execute a query that displays a single summary value for all the rows**
 - Create a query that will return the average vacation hours and the total sick leave hours for Vice Presidents.
 - Use the **AVG** and **SUM** aggregate functions.
 - Execute the query and review the results.
- ▶ **Task 3: Create and execute a second query that displays a single summary value for all the rows**
 - Create a query that returns a count of employees.
 - Use the **COUNT** function.
 - Execute the query and review the results.
- ▶ **Task 4: Create and execute a third query that computes the total number of employees with the Addressline2 value as NULL**
 - Create a query that returns a count of employees where the **AddressLine2** column has **NULL** values.
 - Use the **COUNT** and **ISNULL** functions.
 - Execute the query and review the results.

► **Task 5: Modify the query to eliminate the NULL values**

- Create a query that returns the same values as Task 4 but without taking NULL values into account.
- Use the **COUNT** function:
 - Remember that the **COUNT** function when used with a column name will ignore NULL values.
- Execute the query and review the results.

Results: After this exercise, you should have launched SQL Server Management Studio and created queries to display summary values for rows. You should have also created queries to compute totals and eliminate NULL values.

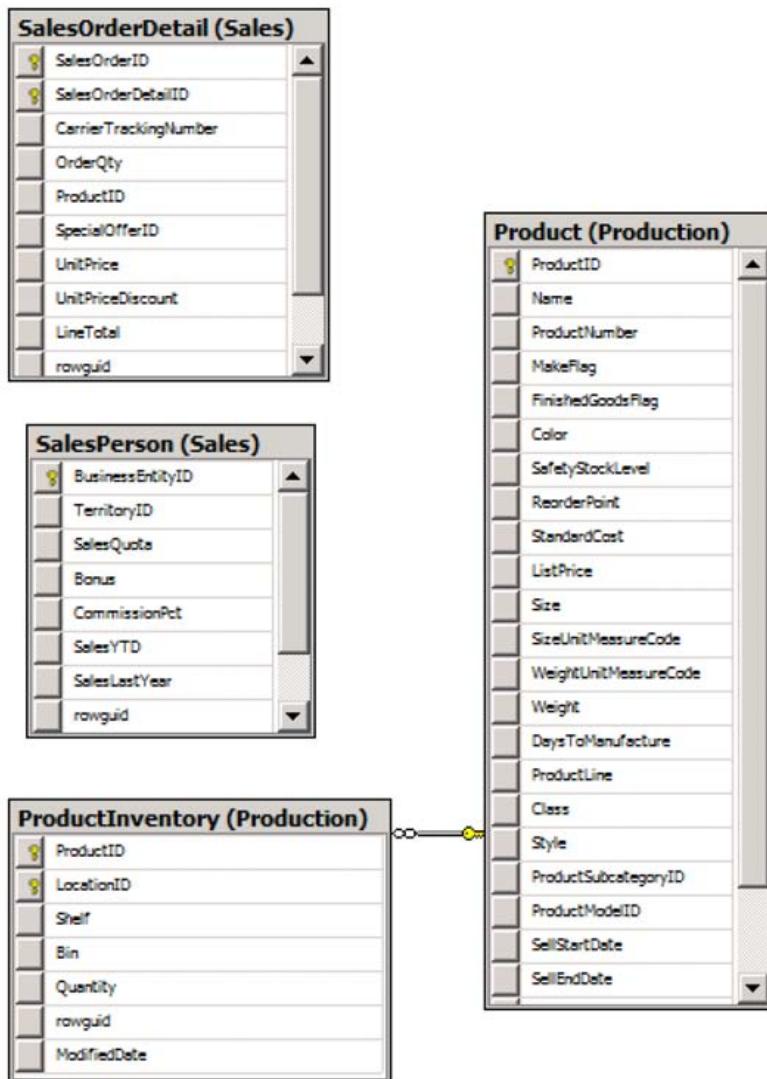
Exercise 2: Summarizing Grouped Data

Scenario

The warehouse manager requires three reports on the average number of hours taken to manufacture a product, the order quantity and the price list of each product, and the safety stock level for red, blue, and black helmets.

This exercise's main tasks are:

1. Create and execute a query to compute the average number of days to manufacture a product.
2. Create and execute a query to display the various colors of a particular product and computes the average ListPrice of the colors.
3. Create and execute a query to generate a report that lists the average order quantity and sum of line total for each product with a line total that exceeds \$1000000.00 and with average quantity less than 3.
4. Create and execute a query to group the products and then compute the sum of the quantity shelf-wise.
5. Create and execute a query to generate a summary report.
6. Distinguish summary and aggregated data from actual table rows.
7. Create and execute a query to generate a report of the summary columns by using the GROUPING function.
8. Create and execute a query that generates a report that displays all the products, the unit price, the unit price discount, and the sum of the columns.



- ▶ **Task 1: Create and execute a query that computes the average number of days to manufacture a product**
 - Create a query that returns the product ID and average days to manufacture from the Products table.
 - Use **AVG**.
 - Execute the query and review the results.

- ▶ **Task 2: Create and execute a query that displays the various colors of a particular product and computes the average ListPrice of the colors**
 - Create a query that returns the color and average ListPrice for a specific product.
 - Use a **GROUP BY** to ensure the averages are computed for the correct column.
 - Use a **WHERE** clause to restrict the query to a single product number.
 - Execute the query and review the results.

- ▶ **Task 3: Create and execute a query to generate a report that lists the average order quantity and sum of line total for each product with a line total that exceeds \$1000000.00, and with average quantity less than 3**
 - Create a query that returns the ProductID, average OrderQty and total of LineTotal.
 - Use a **GROUP BY** to ensure the averages are computed for the correct column.
 - Use a **HAVING** clause to restrict the query to run for cases where the total and averages of LineTotal and OrderQty respectively are in the proper ranges.
 - Don't be afraid to use functions in the **HAVING** clause.
 - Execute the query and review the results.
- ▶ **Task 4: Create and execute a query to group the products and then compute the sum of the quantity shelf-wise**
 - Create a query that returns the ProductID Shelf and total Quantity for a group of products.
 - Use a **GROUP BY** with the **ROLLUP** operator to calculate the columns in a specific order.
 - Execute the query and review the results.
- ▶ **Task 5: Create and execute a query to generate a summary report**
 - Create a query that summarizes sales information from the SalesOrderDetail table using the **CUBE** operator.
 - Sum the number of orders.
 - Specify the **CUBE** operator with the correct columns from the table (orders are related to products and the id of the SalesOrderDetail row).
 - Execute the query and review the results.
- ▶ **Task 6: Distinguish the rows generated by the summary or aggregations and actual table rows**
 - Create a query that summarizes sales quotas and total YTD sales from the SalesPerson table using the **CUBE** operator.
 - Use the **GROUPING** operator in tandem with the **CUBE** operator to indicate the column not being aggregated.
 - Sum the SalesYTD column.
 - Execute the query and review the results.
- ▶ **Task 7: Create and execute a query to generate a report of the summary columns by using the GROUPING function**
 - Create a query that summarizes information about the ProductInventory table using the **CUBE** operator and **GROUPING** operator.
 - The CUBE operator should be used with multiple columns and the **GROUPING** should be used with one of those columns.
 - Sum the Quantity column.
 - Execute the query and review the results.

► **Task 8: Create and execute a query to generate a report that displays all the products, the unit price, the unit price discount, and the sum of the columns**

- Create a query to summarize information about the SalesOrderDetail table using the **COMPUTE** clause.
- Sum the UnitPrice and UnitPriceDiscount columns without using aggregate functions and **GROUP BY**
 - use the **COMPUTE** clause.
- Execute the query and review the results.

Results: After this exercise, you have learned how to create queries to compute averages, display additional data and perform summations. You should have also learned how to create queries using aggregation functions and the GROUPING function.

Exercise 3: Ranking Grouped Data

Scenario

The marketing manager wants you to prepare a report on the year-to-date sales of salespersons and rank their names based on their performance.

This exercise's main tasks are:

1. Generate row numbers for rows returned by a query.
2. Create and execute a second query using the RANK function.
3. Create and execute a third query using the DENSE_RANK function.
4. Create and execute a fourth query that sorts data in descending order and group's data into categories.

SalesPerson (Sales)	
BusinessEntityID	
TerritoryID	
SalesQuota	
Bonus	
CommissionPct	
SalesYTD	
SalesLastYear	
rowguid	
ModifiedDate	

ProductInventory (Production)	
ProductID	
LocationID	
Shelf	
Bin	
Quantity	
rowguid	
ModifiedDate	

- ▶ **Task 1: Generate row numbers for each of the rows in the year-to-date sales of the salespersons**
 - Create a query that returns numbered rows for the SalesPerson table.
 - Use the **ROW_NUMBER** function, specifying the ordering.
 - Filter the results to prune away sales persons with null territories and no sales.
 - Execute the query and review the results.
- ▶ **Task 2: Create and execute a second query using the RANK function**
 - Create a query that returns numbered rows for the ProductInventory table.
 - Use the **RANK** function, specifying the ordering.
 - Execute the query and review the results.
- ▶ **Task 3: Create and execute a third query using the DENSE_RANK function**
 - Create a query that returns numbered rows for the ProductInventory table.
 - Use the **DENSE_RANK** function, specifying the ordering.
 - Execute the query and review the results.

► **Task 4: Create and execute a fourth query that sorts data in descending order and group's data into categories**

- Create a query that returns numbered rows for the SalesPerson table.
- Use the **NTILE** function, using the SalesYTD column for the ordering.
- Filter the results to prune away sales persons with null territories and no sales.
- Execute the query and review the results.

Results: After this exercise, you have learned how to create queries that use the various ranking functions to organize and rank grouped data.

Exercise 4: Creating Crosstab Queries

In this exercise, you will use built in aggregate functions to summarize data in different ways. You will also work around NULL values in your summaries.

This exercise's main tasks are:

1. Create and execute a query using **SELECT** and the **PIVOT** operator.
2. Create and execute a second query using **SELECT** and the **UNPIVOT** operator.

Product (Production)	
ProductID	
Name	
ProductNumber	
MakeFlag	
FinishedGoodsFlag	
Color	
SafetyStockLevel	
ReorderPoint	
StandardCost	
ListPrice	
Size	
SizeUnitMeasureCode	
WeightUnitMeasureCode	
Weight	
DaysToManufacture	
ProductLine	
Class	
Style	
ProductSubcategoryID	
ProductModelID	
SellStartDate	
SellEndDate	
DiscontinuedDate	
rowguid	
ModifiedDate	

► **Task 1: Create and execute a query using SELECT and the PIVOT operator**

- Create a query that returns a pivoted table expression from the `Product` table.
- Use the **PIVOT** operator.
- Filter the input to only use "helmet" products.
- Sum the amount of each helmet by the colors of Red, Blue and Black.
- Order the output by product name.
- Execute the query and review the results.

► **Task 2: Create and execute a second query using SELECT and the UNPIVOT operator**

- Create a query that uses the **UNPIVOT** operator to turn a table expression into column values.
- Execute the query and review the results.

Results: After this exercise, you have learned how to use the PIVOT operator to create a crosstab query and also how to use the UNPIVOT operator.

Lab Shutdown

After you complete the lab, you must shut down the **2778A-NY-SQL-01** virtual machine and discard any changes.

Module Review and Takeaways

• Review Questions

Review Questions

1. When would you use a HAVING clause and not a WHERE clause?
2. What are the differences between COMPUTE and SUM?
3. If you wanted to group products by the average price of the product into four equal groups, what function would you use?
4. How many ranking functions are there?

Module 4

Joining Data from Multiple Tables

Contents:

Lesson 1: Querying Multiple Tables by Using Joins	4-3
Lesson 2: Applying Joins for Typical Reporting Needs	4-11
Lesson 3: Combining and Limiting Result Sets	4-17
Lab: Joining Data from Multiple Tables	4-24

Module Overview

- **Querying Multiple Tables by Using Joins**
- **Applying Joins for Typical Reporting Needs**
- **Combining and Limiting Result Sets**

Joins in Microsoft® SQL Server® 2008 allow you to retrieve data from two or more tables based on logical relationships between those tables. Joins indicate how SQL Server uses data from one table to select the rows in another table. In this module, you will learn how to query multiple tables by using joins, how to apply joins to typical reporting needs, and how to combine and limit join result sets.

Lesson 1

Querying Multiple Tables by Using Joins

- Fundamentals of Joins
- Categorizing Statements by Types of Joins
- Joining Data Using Inner Joins
- Joining Data Using Outer Joins
- Joining Data Using Cross Joins
- Identifying the Potential Impact of a Cartesian Product

A join condition defines the way two tables are related in query by specifying the column from each table to be used for the join and by specifying a logical operator to be used in comparing values from those columns. In this lesson, you will be introduced to the concepts of joins including what joins are and what they do, categorizing the types of joins, and joining data using inner, outer, and cross joins.

Fundamentals of Joins

Joins:

- **Select Specific Columns from Multiple Tables**
 - JOIN keyword specifies that tables are joined and how to join them
 - ON keyword specifies join condition
- **Query Two or More Tables to Produce a Result Set**
 - Use Primary and Foreign Keys as join conditions
 - Use columns common to specified tables to join tables

Simplified JOIN Syntax:

```
FROM first_table join_type second_table  
[ON (join_condition)]
```

Key Points

Join conditions can be specified in either the FROM or WHERE clauses; specifying them in the FROM clause is recommended. WHERE and HAVING clauses can also contain search conditions to further filter the rows selected by the join conditions.

- The tables or views in the FROM clause can be specified in any order with an inner join or full outer join.
- Specifying the join conditions in the FROM clause helps separate them from any other search conditions that may be specified in a WHERE clause, and is the recommended method for specifying joins.
- Joins can be categorized as inner joins, outer joins, or cross joins.

Categorizing Statements by Types of Joins

- Inner Join
 - Includes equi-joins and natural joins
 - Use comparison operators to match rows
- Outer Join
 - Includes left, right, or full outer joins
- Cross Join
 - Also called Cartesian products
- Self Join
 - Refers to any join used to join a table to itself

Key Points

Joins can be categorized as:

- Inner joins that use a comparison operator to match rows from two tables based on the values in common columns from each table.
- Outer joins which can be left, right, or full outer joins.
 - The result set of a left outer join includes all the rows from the left table specified in the LEFT OUTER clause, not just the ones in which the joined columns match.
 - A right outer join is the reverse of a left outer join. All rows from the right table are returned.
 - A full outer join returns all rows in both the left and right tables.
- Cross joins that return all rows from the left table. Each row from the left table is combined with all rows from the right table. Cross joins are also called Cartesian products.

Joining Data Using Inner Joins

- An inner join is a join in which the values in the columns being joined are compared using a comparison operator

Example:

```
SELECT e.LoginID  
FROM HumanResources.Employee AS e  
INNER JOIN Sales.SalesPerson AS s  
ON e.BusinessEntityID = s.BusinessEntityID
```

Result Set:

LoginID

adventure-works\syed0
adventure-works\david8
adventure-works\garrett1
...
(17 row(s) affected)



Key Points

An inner join is a join in which the values in the columns being joined are compared using a comparison operator.

- In the ISO standard, inner joins can be specified in either the FROM or WHERE clause. This is the only type of join that ISO supports in the WHERE clause. Inner joins specified in the WHERE clause are known as old-style inner joins.
- An inner join joins all the columns in both tables, and returns only the rows for which there is an equal value in the join column is known as an equi-join.
- You can also join values in two columns that are not equal. The same operators and predicates used for inner joins can be used for not-equi-joins which will be covered later in this module.

Joining Data Using Outer Joins

- Outer Joins return all rows from at least one of the tables or views mentioned in the FROM clause

Example:

```
SELECT p.Name, pr.ProductReviewID
FROM Production.Product p
LEFT OUTER JOIN
Production.ProductReview pr
ON p.ProductID = pr.ProductID
```

Result Set:

Name	ProductReviewID
Adjustable Race	NULL
Bearing Ball	NULL
...	
(505 row(s) affected)	



Key Points

Outer joins return all rows from at least one of the tables or views mentioned in the FROM clause, as long as those rows meet any WHERE or HAVING search conditions.

- All rows are retrieved from the left table referenced with a left outer join.
- All rows from the right table are referenced in a right outer join.
- All rows from both tables are returned in a full outer join.
- To retain the non-matching information by including non-matching rows in the results of a join, use a full outer join. SQL Server provides the full outer join operator, FULL OUTER JOIN, which includes all rows from both tables, regardless of whether or not the other table has a matching value.

Joining Data Using Cross Joins

- In a Cross Join, each row from the left table is combined with all rows from the right table

Example:

```
SELECT p.BusinessEntityID, t.Name AS Territory
FROM Sales.SalesPerson p
CROSS JOIN Sales.SalesTerritory t
ORDER BY p.BusinessEntityID
```

Result Set:

BusinessEntityID	Territory
274	Northwest
274	Northeast
...	

(170 row(s) affected)

 **Use CROSS JOINS with caution if you do not need a true Cartesian Product**



Key Points

A Cross Join that does not have a WHERE clause produces the Cartesian product of the tables involved in the join.

- A Cartesian product is defined as all possible combinations of rows in all tables.
- The size of a Cartesian product result set is the number of rows in the first table multiplied by the number of rows in the second table.
- Cross Joins should be used with caution because they can be very resource intensive.

Identifying the Potential Impact of a Cartesian Product

A Cartesian Product:

- Is defined as all possible combinations of rows in all tables**
- Results in a rowset containing the number of rows in the first table times the number of rows in the second**
- Can result in huge result sets that take several hours to complete!**

Key Points

The size of a Cartesian product result set is the number of rows in the first table multiplied by the number of rows in the second table.

- A Cross Join that does not have a WHERE clause produces the Cartesian product of the tables involved in the Join.
- If a WHERE clause is added, the cross join behaves as an inner join.

Demonstration: Querying a Table Using Joins

In this demonstration, you will see how to:

- Query the Table Using an Inner Join
- Query the Table Using an Outer Join
- Query the Table Using a Cross Join

Question: When would it make sense to use an outer join instead of an inner join?

Question: Can you think of any scenarios in which you would use a cross join?

Lesson 2

Applying Joins for Typical Reporting Needs

- Joining Three or More Tables
- Joining a Table to Itself
- Joining Tables by Using Non-Equi Joins
- Joining Tables in a User-Defined Function

Although each join specification joins only two tables, FROM clauses can contain multiple join specifications. This allows many tables to be joined for a single query. In this lesson, you will learn about joining more than two tables, joining a table to itself, joining tables using non-equi joins, and joining a table to a user-defined function.

Joining Three or More Tables

• **FROM clauses can contain multiple Join specifications which allows many tables to be joined in a single Query**

Example:

```
SELECT p.Name, v.Name
FROM Production.Product p
JOIN Purchasing.ProductVendor pv
ON p.ProductID = pv.ProductID
JOIN Purchasing.Vendor v
ON pv.BusinessEntityID = v.BusinessEntityID
WHERE ProductSubcategoryID = 15
ORDER BY v.Name
```

Result Set:

Name	Name
LL Mountain Seat/Saddle	Chicago City Saddles
ML Mountain Seat/Saddle	Chicago City Saddles
...	
(18 row(s) affected)	



Key Points

Although each join specification joins only two tables, FROM clauses can contain multiple join specifications. This allows many tables to be joined for a single query.

- When there is more than one join operator in the same statement, either to join more than two tables or to join more than two pairs of columns, the join expressions can be connected with AND or with OR.
- In the case of the ProductVendor table of the AdventureWorks database, joining three or more tables allows you to retrieve data showing the links between the Product, ProductVendor, and Vendor tables to better determine which products are sold by which vendors.
 - One of the tables in the FROM clause, ProductVendor, does not contribute any columns to the results.
 - None of the joined columns, ProductID and VendorID, appear in the results.
 - The middle table of the join, the ProductVendor table, can be called the translation table or intermediate table, because ProductVendor is an intermediate point of connection between the other tables involved in the join.

Joining a Table to Itself

• A Table can be Joined to itself by using a Self-Join

Example:

```
SELECT DISTINCT pv1.ProductID, pv1.BusinessEntityID
FROM Purchasing.ProductVendor pv1
INNER JOIN Purchasing.ProductVendor pv2
ON pv1.ProductID = pv2.ProductID
AND pv1.BusinessEntityID <> pv2.BusinessEntityID
ORDER BY pv1.ProductID
```

Result Set:

ProductID	BusinessEntityID
317	1578
317	1678
...	

(347 row(s) affected)



Key Points

A table can be joined to itself in a self-join. For example, you could use a self-join to find the products that are supplied by more than one vendor in order to better determine which vendor to purchase those products from.

- A self-join involves joining a table to itself by using the same table twice in the query but distinguishing each instance of the table with aliases.
- In the example shown here, pv1 and pv2 are aliases for the Purchasing.ProductVendor table. These aliases are used to qualify the column names in the rest of the query.

Joining Tables by Using Non-Equi Joins

- The same Operators and Predicates used for Inner Joins can be used for Not-Equal Joins

Example:

```
SELECT DISTINCT p1.ProductSubcategoryID, p1.ListPrice
FROM Production.Product p1
INNER JOIN Production.Product p2
ON p1.ProductSubcategoryID = p2.ProductSubcategoryID
AND p1.ListPrice <> p2.ListPrice
WHERE p1.ListPrice < $15 AND p2.ListPrice < $15
ORDER BY ProductSubcategoryID
```

Result Set:

ProductSubcategoryID	ListPrice
23	8.99
23	9.50
...	
(8 row(s) affected)	



Key Points

You can join values in two columns that are not equal.

- The same operators and predicates used for inner joins can be used for not-equal joins.
- The not-equal join ($<\gt;$) is rarely used.
- As a general rule, not-equal joins make sense only when used with a self-join.
- Non-equal joins can be useful for example on the ProductVendor table of the AdventureWorks database to determine which products have more than one vendor.

Joining Tables in a User-Defined Function

• **User-defined functions can be used to focus, simplify, and customize the perception each user has of the database**

Example:

```
CREATE FUNCTION Sales.ufn_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 );
```

Result Set:

Product ID	Name	YTD Total
707	Sport-100 Helmet, Red	620.250910
708	Sport-100 Helmet, Black	657.636610



Key Points

Inline user-defined functions are a subset of user-defined functions that return a table data type.

- Inline user-defined functions follow these rules:
 - The RETURNS clause contains only the keyword table. You do not define the format of a return variable because it is set by the format of the result set of the SELECT statement in the RETURN clause.
 - There is no function_body delimited by BEGIN and END.
 - The RETURN clause contains a single SELECT statement in parentheses. The result set of the SELECT statement forms the table returned by the function. The SELECT statement used in an inline function is subject to the same restrictions as SELECT statements used in views.
 - The table-valued function accepts only constants or @local_variable arguments.

Demonstration: Joining Tables

In this demonstration, you will see how to:

- Join Three or More Tables
- Join a Table to Itself
- Join a Table using a Non-Equi Join

Question: What is a translation (or intermediate) table and what is it used for?

Question: When does it make sense to use a non-equi join?

Lesson 3

Combining and Limiting Result Sets

- Combining Result Sets by Using the UNION Operator
- Limiting Result Sets by Using the EXCEPT and INTERSECT Operators
- Identifying the Order of Precedence of UNION, EXCEPT, and INTERSECT
- Limiting Result Sets by Using the TOP and TABLESAMPLE Operators
- Categorizing Statements that Limit Result Sets

Many times, you will want to combine or limit result sets to make them clearer. In this lesson, you will learn how to combine result sets by using the UNION operator, how to limit result sets with EXCEPT and INTERSECT, how to identify the order of precedence of UNION, EXCEPT, and INTERSECT, and how to limit result sets with TOP and TABLESAMPLE.

Combining Result Sets by Using the UNION Operator

• UNION combines the results of two or more queries into a single result set that includes all the rows that belong to all queries in the union

Example:

```
SELECT * FROM testa  
UNION ALL  
SELECT * FROM testb;
```

Result Set:

columna	columnb
100	test
100	test
...	

(8 row(s) affected)

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



Key Points

UNION combines the results of two or more queries into a single result set that includes all the rows that belong to all queries in the union. Using a UNION query can be useful for example when you simply need to retrieve all of the data from two tables and do not actually need to join the data in any way.

- The following are basic rules for combining the result sets of two queries by using UNION:
 - The number and the order of the columns must be the same in both queries.
 - The data types must be compatible in both queries.
- When the data types are the same but differ in precision, scale, or length, the result is determined based on the same rules for combining expressions.

Limiting Result Sets by Using the EXCEPT and INTERSECT Operators

- EXCEPT returns any distinct values from the query to the left of the EXCEPT operand that are not also returned from the right query
- INTERSECT returns any distinct values that are returned by both the query on the left and right sides of the INTERSECT operand

EXCEPT Example:

```
SELECT ProductID  
FROM Production.Product  
EXCEPT  
SELECT ProductID  
FROM Production.WorkOrder
```

Result Sets

ProductID	-----
429	
...	
(266 row(s) affected)	

INTERSECT Example:

```
SELECT ProductID  
FROM Production.Product  
INTERSECT  
SELECT ProductID  
FROM Production.WorkOrder
```

Result Sets

ProductID	-----
3	
...	
(238 row(s) affected)	

Key Points

You can use the EXCEPT and INTERSECT operators to compare the results of two or more SELECT statements and return distinct values.

- EXCEPT returns any distinct values from the left query that are not also found on the right query.
- INTERSECT returns any distinct values that are returned by both the queries on the left and right sides of the INTERSECT operand.
- The basic rules for combining the result sets of two queries that use EXCEPT or INTERSECT are the following:
 - The number and the order of the columns must be the same in all queries.
 - The data types must be compatible.

Identifying the Order of Precedence of UNION, EXCEPT, and INTERSECT

EXCEPT, INTERSECT, and UNION are evaluated in the context of the following precedence:

1 Expressions in parentheses

2 The INTERSECT operand

3 EXCEPT and UNION evaluated from Left to Right based on their position in the expression

Key Points

If EXCEPT or INTERSECT is used together with other operators in an expression, it is evaluated in the context of the following precedence:

- Expressions in parentheses
- The INTERSECT operand
- EXCEPT and UNION evaluated from left to right based on their position in the expression

If EXCEPT or INTERSECT is used to compare more than two sets of queries, data type conversion is determined by comparing two queries at a time, and following the previously mentioned rules of expression evaluation.

Limiting Result Sets by Using the TOP and TABLESAMPLE Operators

• **TOP and TABLESAMPLE limit the number of rows returned in a result set**

Result Sets

TOP Example:

```
SELECT TOP (15)
    FirstName, LastName
FROM Person.Person
```

TABLESAMPLE Example:

```
SELECT
    FirstName, LastName
FROM Person.Person
TABLESAMPLE (1 PERCENT)
```

FirstName	LastName
Syed	Abbas
Catherine	Abel
...	
(15 row(s) affected)	

FirstName	LastName
Eduardo	Barnes
Edward	Barnes
...	
(199 row(s) affected)	

Key Points

If a SELECT statement that includes TOP also has an ORDER BY clause, the rows to be returned are selected from the ordered result set.

- TABLESAMPLE cannot be applied to derived tables, tables from linked servers, and tables derived from table-valued functions, rowset functions, or OPENXML.
- TABLESAMPLE can be used to quickly return a sample from a large table when either of the following conditions is true:
 - The sample does not have to be a truly random sample at the level of individual rows.
 - Rows on individual pages of the table are not correlated with other rows on the same page.

Question: When would you use TOP to limit a result set?

Question: When would you use TABLESAMPLE to limit a result set?

Categorizing Statements That Limit Result Sets

- **UNION**
 - Combines the results of two or more SELECT statements into a single result set
- **EXCEPT and INTERSECT**
 - Compares the results of two or more SELECT statements and return distinct values
- **TOP**
 - Specifies that only the first set of rows will be returned from the query result
- **TABLESAMPLE**
 - Limits the number of rows returned from a table in the FROM clause to a sample number or PERCENT of rows

Key Points

We can categorize statements that limit result sets into four types.

- UNION specifies that multiple result sets are to be combined and returned as a single result set.
- EXCEPT returns any distinct values from the query to the left of the EXCEPT operand that are not also returned from the right query.
- INTERSECT returns any distinct values that are returned by both the query on the left and right sides of the INTERSECT operand.
- The TOP and TABLESAMPLE statements can use either a number or a percent of the rows.

Demonstration: Combining and Limiting Result Sets

In this demonstration, you will see how to:

- Combine Result Sets
- Limit Result Sets using TABLESAMPLE
- Limit Result Sets using TOP

Question: What are the basic rules for combining the result sets of two queries by using UNION?

Question: How do you think rows are returned from a SELECT TOP statement that also has an ORDER BY clause?

Question: What conditions must be met in order to use TABLESAMPLE to return a sample from a large table?

Lab: Joining Data from Multiple Tables

- Exercise 1: Querying Multiple Tables by Using Joins
- Exercise 2: Applying Joins for Typical Reporting Needs
- Exercise 3: Combining and Limiting Result Sets

Logon information

Virtual machine	NY-SQL-01
User name	Administrator
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Exercise 1: Querying Multiple Tables by Using Joins

Scenario

You are the database developer of Adventure Works. You have been asked by the various managers to prepare the following reports:

- List of employee login IDs for those employees who are also salespeople.
- List of products produced by the company, regardless of whether those products have reviews written about them or not.
- List of all sales persons regardless of whether or not they are assigned a sales territory.

The main tasks for this exercise are as follows:

1. Launch SQL Server Management Studio.
2. Create and execute an Inner Join.
3. Create and execute a Left Outer Join and a Right Outer Join.

► Task 1: Launch SQL Server Management Studio

- Start the 2778A-NY-SQL-01 virtual machine, and log on as **Administrator** with the password of **Pa\$\$w0rd**.
- In Microsoft® Windows® Explorer, browse to E:\MOD04\Labfiles\Starter and run LabSetup.cmd.
- Start SQL Server Management Studio.

► Task 2: Create and execute an Inner Join

- Create a query that will return the LoginID column based on the BusinessEntityID.
- Use an Inner Join to join the HumanResources.Employee table with the Sales.SalesPerson table.

- Execute the query and review the results.

► **Task 3: Create and execute a Left Outer Join and a Right Outer Join**

- Create a Left Outer Join that uses the Product and ProductReview tables on their ProductID columns to show only those products for which reviews have been written.
- Create a Right Outer Join that uses the SalesTerritory and SalesPerson tables on their TerritoryID columns to show any territory that has been assigned to a salesperson.
- Execute the query and review the results.

Results: After this exercise, you should have launched SQL Server Management Studio and created and executed an Inner Join. You should have also created and executed a Left Outer Join and a Right Outer Join.

Exercise 2: Applying Joins for Typical Reporting Needs

Scenario

You have now been asked by the various managers of Adventure Works to prepare the following reports:

- List of all products that are produced by vendors.
- List of products that are created by all vendors, eliminating duplicates using the DISTINCT operator.
- List of product subcategories that have at least two different prices less than \$15.

The main tasks for this exercise are as follows:

1. Create and execute a query using a Self Join.
2. Create and execute a query using a Self Join and the DISTINCT clause.
3. Create and execute a Non-Equi Join with an equality and a non-equality operator.

► **Task 1: Create and execute a query using a Self Join**

- Create a query that will return a list of all products on the Purchasing.ProductVendor table that are produced by all vendors.
- Use a Self Join and order the results by ProductID.
- Execute the query and review the results.

► **Task 2: Create and execute a query using a Self Join and the DISTINCT clause**

- Create a query that will return a list of all products on the Purchasing.ProductVendor table that are produced by vendors.
- Use a Self Join that uses the DISTINCT clause, and order the results by ProductID.
- Execute the query and review the results.

► **Task 3: Create and execute a Non-Equi Join with an equality and a non-equality operator**

- Create a query that will return a list of all product subcategories that have at least two different prices less than \$15.
- Use an Inner Join to join the Production.Product table to itself and a non-equality operator on the ListPrice column to select two different prices.
- Execute the query and review the results.

Results: After this exercise, you should have created and executed a query using a self join, created and executed a query using a self join and the DISTINCT clause, and created and executed a non-equi join with both an equality and non-equality operator.

Exercise 3: Combining and Limiting Result Sets

Scenario

You have been asked by the various managers of Adventure Works to prepare the following reports:

- Combined list of table TestA and table TestB using the UNION ALL operator.
- List of all product IDs from the Production.Product table that are not on the Production.WorkOrder table.
- List of all product IDs with distinct values on both the Production.Product table and Production.WorkOrder table.
- List of the top 15% of products from the Production.Product table, ordered by ProductID.
- Randomly generated list of 10% of people from the Person.Person table including first name and surname.

The main tasks for this exercise are as follows:

1. Combine the result sets of two queries by using the UNION ALL operator.
2. Limit result sets by using the EXCEPT clause with the SELECT statement.
3. Limit result sets by using the INTERSECT clause with the SELECT statement.
4. Limit result sets by using the TOP and TABLESAMPLE operators.

► **Task 1: Combine the result sets of two queries by using the UNION ALL operator**

- Create a query that will return a combined list of all columns on the TestA and TestB tables.
- Do not limit the results of the query with any operators.
- Execute the query and review the results.

► **Task 2: Limit result sets by using the EXCEPT clause with the SELECT statement**

- Create a query using EXCEPT that lists all product IDs from the Production.Product table that are not on the Production.WorkOrder table.
- Execute the query and review the results.

► **Task 3: Limit result sets by using the INTERSECT clause with the SELECT statement**

- Create a query using INTERSECT that lists all product IDs from the Production.Product table that are not on the Production.WorkOrder table.
- Execute the query and review the results.

► **Task 4: Limit result sets by using the TOP and TABLESAMPLE operators**

- Create a query that lists of the top 15% of products from the Production.Product table, ordered by ProductID.
- Execute the query and review the results.
- Create a query that randomly generates a list of 10% of people from the Person.Person table including first name and surname.
- Execute the query and review the results.

Results: After this exercise, you should have combined the result sets of two queries by using the UNION ALL operator. You should have also limited result sets by using the EXCEPT and INTERSECT clauses. Finally, you should have limited result sets using the TOP and TABLESAMPLE operators.

Lab Shutdown

After you complete the lab, you must shut down the 2778A-NY-SQL-01 virtual machine and discard any changes.

Module Review and Takeaways

- Review Questions
- Best Practices

Review Questions

1. How does a join condition define the way two tables are related in a query?
2. How can multiple join operators be combined in the same statement?
3. What options are available to the TABLESAMPLE clause and what are they used for?

Best Practices related to using Joins

Supplement or modify the following best practices for your own work situations:

- Specifying the join conditions in the FROM clause helps separate them from any other search conditions that may be specified in a WHERE clause, and is the recommended method for specifying joins.
- Columns used in a join condition are not required to have the same name or be the same data type. However, if the data types are not identical, they must be compatible, or be types that SQL Server can implicitly convert.
- To retain the non-matching information by including non-matching rows in the results of a join, use a full outer join. SQL Server provides the full outer join operator, FULL OUTER JOIN, which includes all rows from both tables, regardless of whether or not the other table has a matching value.

Best Practices related to using the UNION operator

Supplement or modify the following best practices for your own work situations:

- All select lists in the statements that are being combined with UNION must have the same number of expressions (column names, arithmetic expressions, aggregate functions, and so on).
- Corresponding columns in the result sets that are being combined with UNION, or any subset of columns used in individual queries, must be of the same data type, have an implicit data conversion possible between the two data types, or have an explicit conversion supplied. For example, a UNION between a column of datetime data type and one of binary data type will not work unless an explicit

conversion is supplied. However, a UNION will work between a column of money data type and one of int data type, because they can be implicitly converted.

- Corresponding result set columns in the individual statements that are being combined with UNION must occur in the same order, because UNION compares the columns one-to-one in the order given in the individual queries.

Best Practices related to using the EXCEPT and INTERSECT operators

Supplement or modify the following best practices for your own work situations:

- The definitions of the columns that are part of an EXCEPT or INTERSECT operation do not have to be the same, but they must be comparable through implicit conversion.
- The query specification or expression cannot return xml, text, ntext, image, or nonbinary CLR user-defined type columns because these data types are not comparable.

Module 5

Working with Subqueries

Contents:

Lesson 1: Writing Basic Subqueries	5-3
Lesson 2: Writing Correlated Subqueries	5-10
Lesson 3: Comparing Subqueries with Joins and Temporary Tables	5-16
Lesson 4: Using Common Table Expressions	5-20
Lab: Working with Subqueries	5-25

Module Overview

- Writing Basic Subqueries
- Writing Correlated Subqueries
- Comparing Subqueries with Joins and Temporary Tables
- Using Common Table Expressions

A subquery in Microsoft® SQL Server® 2008 is a query that is nested inside a statement or another subquery. Common Table Expressions (CTE) can be thought of as temporary result sets. In this module, you will learn about writing basic subqueries and writing correlated subqueries. You will also compare subqueries with joins and temporary tables. Finally, you will see how to use Common Table Expressions.

Lesson 1

Writing Basic Subqueries

- What Are Subqueries?
- Using Subqueries as Expressions
- Using the ANY, ALL, and SOME Operators
- Scalar versus Tabular Subqueries
- Rules for Writing Subqueries

A subquery is a query that is nested inside a SELECT, INSERT, UPDATE, or DELETE statement, or inside another subquery that can be used anywhere an expression is allowed. In this lesson, you will learn how subqueries work. You will learn about using subqueries as expressions and using the ANY, ALL, and SOME operators. You will also discuss the differences between scalar and tabular subqueries and finally you will review considerations for writing basic subqueries.

What Are Subqueries?

Queries nested inside a SELECT, INSERT, UPDATE, or DELETE statement

Can be used anywhere an Expression is allowed

Example:

```
SELECT ProductID, Name
FROM Production.Product
WHERE Color NOT IN
    (SELECT Color
     FROM Production.Product
     WHERE ProductID = 5)
```

Result Set:

ProductID	Name
1	Adjustable Race
2	Bearing Ball
...	

(504 row(s) affected)

Key Points

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.
- 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.
- The subquery shown here is used as a column expression named MaxUnitPrice in a SELECT statement.

Using Subqueries as Expressions

A Subquery can be substituted anywhere an expression can be used in the following statements, except in an ORDER BY list:

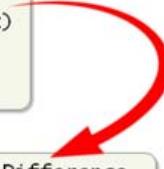
<input checked="" type="checkbox"/> SELECT	<input checked="" type="checkbox"/> INSERT
<input checked="" type="checkbox"/> UPDATE	<input checked="" type="checkbox"/> DELETE

Example:

```
SELECT Name, ListPrice,
       (SELECT AVG(ListPrice) FROM Production.Product)
        AS Average, ListPrice -
       (SELECT AVG(ListPrice) FROM Production.Product)
        AS Difference
  FROM Production.Product
 WHERE ProductSubcategoryID = 1
```

Result Set:

Name	ListPrice	Average	Difference
Mountain-100 Silver, 38	3399.99	438.6662	2961.3238
Mountain-100 Silver, 42	3399.99	438.6662	2961.3238
...			
(32 row(s) affected)			



Key Points

Subqueries can be specified in many places, including in place of an expression.

- A subquery can be substituted anywhere an expression can be used in SELECT, UPDATE, INSERT, and DELETE statements, except in an ORDER BY list.
- The query shown here finds the prices of all mountain bike products, their average price, and the difference between the price of each mountain bike and the average price.

Using the ANY, ALL, and SOME Operators

- Comparison operators that introduce a subquery can be modified by the keywords ALL or ANY
- SOME is an ISO standard equivalent for ANY

ANY Example:

```
SELECT Name
FROM Production.Product
WHERE ListPrice >= ANY
  (SELECT MAX (ListPrice)
   FROM Production.Product
   GROUP BY ProductSubcategoryID)
```

Result Sets

Name

LL Mountain Seat
ML Mountain Seat ...
(304 row(s) affected)

ALL Example:

```
SELECT Name
FROM Production.Product
WHERE ListPrice >= ALL
  (SELECT MAX (ListPrice)
   FROM Production.Product
   GROUP BY ProductSubcategoryID)
```

Result Sets

Name

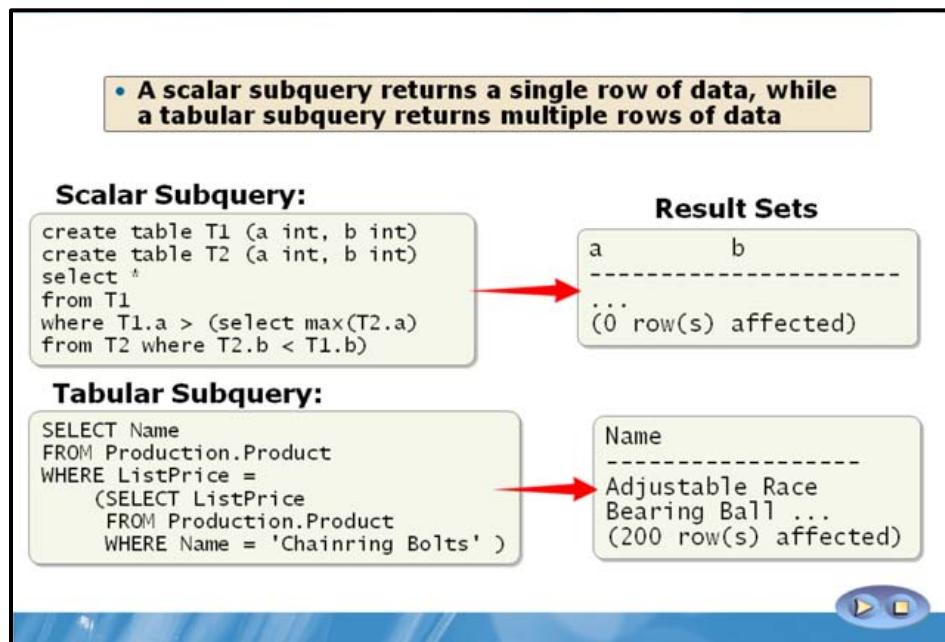
Road-150 Red, 62
Road-150 Red, 44...
(5 row(s) affected)

Key Points

Comparison operators that introduce a subquery can be modified by the keywords ALL or ANY. SOME is an ISO standard equivalent for ANY.

- ALL compares a scalar value with a single-column set of values and returns **TRUE** when the comparison specified is TRUE for all pairs (scalar_expression, x), where x is a value in the single-column set; otherwise, ALL returns **FALSE**.
- SOME and ANY also compare a scalar value with a single-column set of values, but return **TRUE** when the comparison specified is TRUE for any pair (scalar_expression, x) where x is a value in the single-column set; otherwise, each would return **FALSE**.
- Using the `>` comparison operator as an example, `>ALL` means greater than every value. In other words, it means greater than the maximum value. For example, `>ALL (1, 2, 3)` means greater than 3. `>ANY` means greater than at least one value, that is, greater than the minimum. So `>ANY (1, 2, 3)` means greater than 1.
- For a row in a subquery with `>ALL` to satisfy the condition specified in the outer query, the value in the column introducing the subquery must be greater than each value in the list of values returned by the subquery.

Scalar versus Tabular Subqueries



Key Points

- A scalar subquery is a subquery that returns a single row.
- A tabular subquery is a subquery that returns tabular data.
- The subquery in this example uses an aggregate that ensures that it produces only a single row on each execution.

Rules for Writing Subqueries

Subquery Allowances <ul style="list-style-type: none">• Subqueries can be specified in many places• A subquery can itself include one or more subqueries
Subquery Restrictions <ul style="list-style-type: none">• SELECT list of a subquery introduced with a comparison operator can include only one expression• WHERE clauses must be join-compatible• ntext, text, and image data types cannot be used• Column names in a statement are implicitly qualified by the table referenced in the FROM clause

Key Points

Writing subqueries requires that you have an understanding of the rules that subqueries follow.

- Subqueries can be specified in many places including:
 - With aliases.
 - With IN or NOT IN.
 - In UPDATE, DELETE, and INSERT statements.
 - With comparison operators.
 - With ANY, SOME, or ALL.
 - With EXISTS or NOT EXISTS.
 - In place of an expression.
- Subqueries have several restrictions imposed upon them including:
 - The select list of a subquery introduced with a comparison operator can include only one expression or column name.
 - If the WHERE clause of an outer query includes a column name, it must be join-compatible with the column in the subquery select list.
 - The **ntext**, **text**, and **image** data types cannot be used in the select list of subqueries.
 - The general rule in qualifying column names is that column names in a statement are implicitly qualified by the table referenced in the FROM clause at the same level.
- A subquery can itself include one or more subqueries and any number of subqueries can be nested in a statement.

Demonstration: Writing Basic Subqueries

In this demonstration, you will see how to:

- Write a Basic Subquery
- Use the ANY, ALL, and SOME Operators

Question: How would you rewrite the basic subquery demonstrated to you here as a join?

Question: What results were returned by the SELECT statement that contained an ANY expression and why?

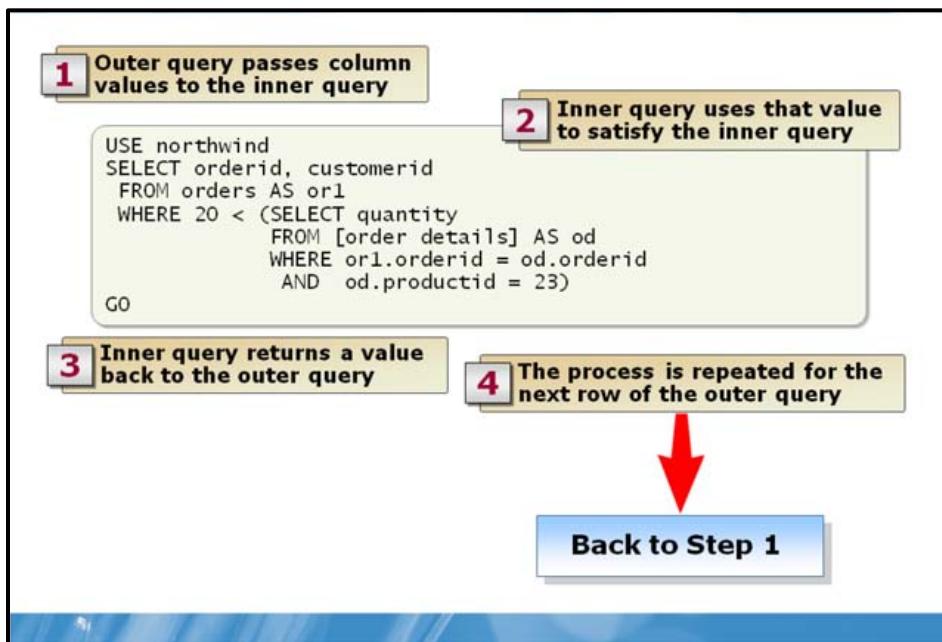
Lesson 2

Writing Correlated Subqueries

- What Are Correlated Subqueries?
- Building a Correlated Subquery
- Using Correlated Subqueries
- Using the EXISTS Clause with Correlated Subqueries

Many queries can be evaluated by executing the subquery once and substituting the resulting value or values into the WHERE clause of the outer query, turning the subquery into a correlated, or repeating, subquery. In this lesson, you will learn how correlated subqueries work. You will also discuss how to build a correlated subquery, writing repeating queries by using correlated subqueries, and using the EXISTS clause with correlated subqueries.

What Are Correlated Subqueries?

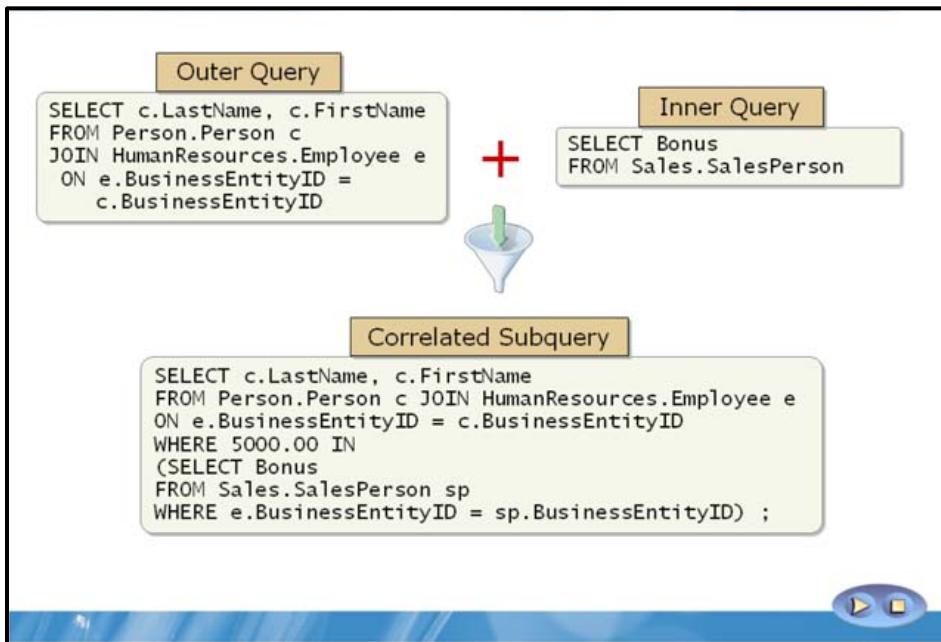


Key Points

A correlated subquery is a subquery that contains a reference to a column or columns in the outer query. Correlated subqueries are also called repeating subqueries because they are executed multiple times during processing—one time for each row in the outer query.

- In queries that include a correlated subquery (also known as a repeating subquery), the subquery depends on the outer query for its values. This means that the subquery is executed repeatedly, once for each row that might be selected by the outer query.
- Correlated subqueries can also include table-valued functions in the FROM clause by referencing columns from a table in the outer query as an argument of the table-valued function. In this case, for each row of the outer query, the table-valued function is evaluated according to the subquery.

Building a Correlated Subquery



Key Points

- A subquery depends on the outer query for its values. Because of this, building a correlated subquery involves creating both an outer and inner query.
- The correlated subquery shown here retrieves one instance of each employee's first and last name for which the bonus in the SalesPerson table is 5000 and for which the employee identification numbers match in the Employee and SalesPerson tables.

Using Correlated Subqueries

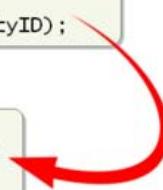
- **Correlated subqueries are executed repeatedly, once for each row that may be selected by the outer query**

Example:

```
SELECT DISTINCT c.LastName, c.FirstName  
FROM Person.Person c JOIN HumanResources.Employee e  
ON e.BusinessEntityID = c.BusinessEntityID  
WHERE 5000.00 IN  
(SELECT Bonus  
FROM Sales.SalesPerson sp  
WHERE e.BusinessEntityID = sp.BusinessEntityID);
```

Result Set:

LastName	FirstName
Ansmann-Wolfe	Pamela
Saraiva	José
(2 row(s) affected)	



Key Points

- The query shown here depends on the outer query for its values. The query is executed repeatedly, one time for each row that may be selected by the outer query.
- The query retrieves one instance of the first and last name of each employee for which the bonus in the SalesPerson table is 5000.00 and for which the employee identification numbers match in the Employee and SalesPerson tables.
- SQL Server considers each row of the Employee table for inclusion in the results by substituting the value in each row into the inner query.

Using the EXISTS Clause with Correlated Subqueries

- When a subquery is introduced with the keyword EXISTS, the subquery functions as an existence test

Example:

```
SELECT Name
FROM Production.Product
WHERE EXISTS
    (SELECT * FROM Production.ProductSubcategory
     WHERE ProductSubcategoryID =
Production.Product.ProductSubcategoryID
     AND Name = 'Wheels')
```

Result Set:

Name
LL Mountain Front Wheel
ML Mountain Front Wheel
...
(14 row(s) affected)



Key Points

Subqueries introduced with the EXISTS clause do not actually produce any data, instead they return a value of TRUE or FALSE.

- The EXISTS operator is used to test for the existence of rows, and returns TRUE if a subquery contains any rows.
- The keyword EXISTS is not preceded by a column name, constant, or other expression.
- The select list of a subquery introduced by EXISTS almost always consists of an asterisk (*). There is no reason to list column names because you are just testing whether rows that meet the conditions specified in the subquery exist.

Demonstration: Writing Correlated Subqueries

In this demonstration, you will see how to:

- Write a Correlated Subquery
- Create a Correlated Subquery with Comparison Operators

Question: What is the process behind building a correlated subquery?

Question: What results did the correlated subqueries return?

Lesson 3

Comparing Subqueries with Joins and Temporary Tables

- Subqueries versus Joins
- Temporary Tables
- Subqueries versus Temporary Tables

While joins and temporary tables can sometimes be used in the place of subqueries, subqueries can sometimes produce similar or better results. In this lesson, you will learn the differences between subqueries and joins, discuss the concept of temporary tables, and learn to describe the differences between subqueries and temporary tables.

Subqueries versus Joins

Joins can yield better performance in some cases where existence must be checked

Joins are performed faster by SQL Server than subqueries

Subqueries can often be rewritten as joins

SQL Server 2008 query optimizer is intelligent enough to convert a subquery into a join if it can be done

Subqueries are useful for answering questions that are too complex to answer with joins

Key Points

In Transact-SQL, there is usually no performance difference between a statement that includes a subquery and a semantically equivalent version that does not.

- Joins can yield better performance in some cases where existence must be checked. However, in some cases where existence must be checked, a join yields better performance.
- Subqueries, unlike joins, can operate on lists introduced with IN or EXISTS, and can also operate on lists with a comparison operator modified by ANY or ALL.
- Generally speaking, subqueries are written to answer questions that are too complex to answer with joins or, more appropriately, are easier to write with a subquery than a join. It is often easier to see nesting and query organization in a subquery than it is in a query that requires a number of joins to arrive at the same conclusion—if that query with multiple joins can even duplicate the subquery in the first place.

Temporary Tables

The diagram illustrates the creation of two types of temporary tables:

- Local Temporary Tables:**
 - Have a single number sign (#) as the first character of their names
 - Visible only to the current connection for the user
 - Deleted when the user disconnects from SQL Server
- Global Temporary Tables:**
 - Have a double number sign (##) as the first character of their names
 - Visible to any user once created
 - Deleted when all users referencing them disconnect

Arrows point from the descriptions to their corresponding CREATE TABLE statements:

- Local Temporary Table statement:

```
CREATE TABLE #StoreInfo
(
    EmployeeID int,
    ManagerID int,
    Num int
)
```
- Global Temporary Table statement:

```
CREATE TABLE ##StoreInfo
(
    EmployeeID int,
    ManagerID int,
    Num int
)
```

At the bottom right of the slide are navigation icons for back, forward, and search.

Key Points

Temporary tables are similar to permanent tables, except temporary tables are stored in **tempdb** and are deleted automatically when they are no longer used.

- Local temporary tables are visible only to their creators during the same connection to an instance of SQL Server as when the tables were first created or referenced and are deleted after the user disconnects from SQL Server.
- Global temporary tables are visible to any user and any connection after they are created, and are deleted when all users that are referencing the table disconnect from the instance of SQL Server.

Subqueries versus Temporary Tables

- As subqueries get more complex their performance may decrease
- Maintainability can be easier with subqueries in some situations, and easier with temporary tables in others
- Temporary tables can be easier for some to debug while others prefer to work with a single subquery

Key Points

There are several differences between subqueries and temporary tables that may cause you to use one instead of the other depending on system performance and personal preference.

- Subqueries are appropriate when you have a lot of RAM since subqueries occur in memory.
- Temporary tables are more appropriate when you have a database server or servers with a lot of hard disk space because they require more hard disk resources when executing.
- Subqueries are often easier to maintain than temporary tables. However, if a subquery becomes very complex, you may want to consider breaking it down into multiple temporary table queries in order to filter through smaller chunks of data.

Lesson 4

Using Common Table Expressions

- What Are Common Table Expressions?
- Writing Common Table Expressions
- Writing Recursive Queries by Using Common Table Expressions

A Common Table Expression (CTE) can be thought of as a temporary result set that is defined within the execution scope of a single SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. In this lesson, you will learn how Common Table Expressions work and how to write Common Table Expressions. You will also learn how to write and build recursive queries by using Common Table Expressions.

What Are Common Table Expressions?

The diagram illustrates the three steps to create a Common Table Expression (CTE):

- 1 Choose a CTE name and column list
- 2 Create the CTE SELECT query
- 3 Use the CTE in a query

Below the steps is a sample T-SQL query:

```
WITH TopSales (SalesPersonID, NumSales) AS
    (SELECT SalesPersonID, Count(*)
     FROM Sales.SalesOrderHeader GROUP BY SalesPersonId)

    SELECT LoginID, NumSales
    FROM HumanResources.Employee e INNER JOIN TopSales
    ON TopSales.SalesPersonID = e.EmployeeID
    ORDER BY NumSales DESC
```

At the bottom right of the slide are navigation icons: a blue circle with a white triangle pointing right, a blue circle with a white square, and a blue circle with a white double arrow.

Key Points

A Common Table Expression (CTE) is similar to a derived table since it is not stored as an object and lasts only for the duration of the query.

- A CTE can be used to create a recursive query, substitute for a view when the general use of a view is not required, enable grouping by a column that is derived from a scalar subselect, or reference the resulting table multiple times in the same statement.
- Using a CTE offers the advantages of improved readability and ease in maintenance of complex queries.
- CTEs can be defined in user-defined routines, such as functions, stored procedures, triggers, or views.

Writing Common Table Expressions

1 Choose a CTE name and column list
2 Create the CTE SELECT query
3 Use the CTE in a query

```
WITH TopSales (SalesPersonID, NumSales) AS
    (SELECT SalesPersonID, Count(*)
     FROM Sales.SalesOrderHeader GROUP BY SalesPersonId)

    SELECT LoginID, NumSales
    FROM HumanResources.Employee e INNER JOIN TopSales
    ON TopSales.SalesPersonID = e.EmployeeID
    ORDER BY NumSales DESC
```

Key Points

A CTE is made up of an expression name representing the CTE, an optional column list, and a query defining the CTE.

- After a CTE is defined, it can be referenced like a table or view can in a SELECT, INSERT, UPDATE, or DELETE statement.
- The list of column names as shown by `SELECT <column_list>` above is optional only if distinct names for all resulting columns are supplied in the query definition.
- The example shown shows the components of the CTE structure: expression name, column list, and query.

Writing Recursive Queries by Using Common Table Expressions

Modify CTE SELECT query when creating CTE:

- 1 Create the anchor member query (top of recursion tree)
- 2 Add the UNION ALL operator
- 3 Create the recursive member query that self-references the CTE

```
SELECT ManagerID, EmployeeID
FROM HumanResources.Employee
WHERE ManagerID IS NULL
UNION ALL
SELECT e.ManagerID, e.EmployeeID
FROM HumanResources.Employee e
INNER JOIN HumanResources.Employee mgr
ON e.ManagerID = mgr.EmployeeID
```



Key Points

- A recursive CTE consists of three elements:
 1. Invocation of the routine.
 2. Recursive invocation of the routine.
 3. Termination check.
- Although a recursive routine in other languages returns a scalar value, a recursive CTE can return multiple rows.
- The recursive CTE structure must contain at least one anchor member and one recursive member.

Demonstration: Using Common Table Expressions

In this demonstration, you will see how to:

- Write a Common Table Expression
- Write a Recursive Query

Question: In the first demonstration, how many times was the CTE referenced when the statement was executed and why?

Question: What would happen if you created a recursive CTE that returned the same values for both the parent and child columns?

Lab: Working with Subqueries

- Exercise 1: Writing Basic Subqueries
- Exercise 2: Writing Correlated Subqueries
- Exercise 3: Comparing Subqueries with Joins and Temporary Tables
- Exercise 4: Using Common Table Expressions

Logon information

Virtual machine	NY-SQL-01
User name	Administrator
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Exercise 1: Writing Basic Subqueries

Scenario

You are a database developer of Adventure Works. You have been asked by the Product Manager to prepare the following reports:

- List that contains prices for Chainring Bolts.
- List of all wheel types produced at any time.

The main tasks for this exercise are as follows:

1. Launch **SQL Server Management Studio**.
2. Create a Basic Subquery.
3. Create a Subquery with the EXISTS Keyword.

► Task 1: Launch SQL Server Management Studio

- Start the **2778A-NY-SQL-01** virtual machine, and log on as **Administrator** with the password of **Pa\$\$w0rd**.
- Open **SQL Server Management Studio**.

► Task 2: Create a basic subquery

- Create a subquery to select the name of any products that have a list price equal to the list price of 'Chainring Bolts' on the **Production.Product** table.
- Execute the query and review the results.

► Task 3: Create a subquery with the EXISTS Keyword

- Create subquery with the EXISTS keyword that will return all items of the type 'Wheels' from the **Production.ProductSubcategory** table.

- Execute the query and review the results.

Results: After this exercise, you should have launched SQL Server Management Studio and created and executed a basic subquery. You should have also created and executed a subquery with the EXISTS keyword.

Exercise 2: Writing Correlated Subqueries

Scenario

You have been asked by a Sales Manager to prepare the following reports:

- List of each employee who has received a bonus of over \$5000.
- List of sales where the quantity is less than the average quantity for sales of that product.

The main tasks for this exercise are as follows:

1. Create a Correlated Subquery.
2. Create a Correlated Subquery with Comparison Operators.

► Task 1: Create a correlated subquery

- Create a correlated subquery that returns the last name and first name of each employee who has received a bonus of over \$5000.
- Join the **Person.Person** table with the **HumanResources.Employee** table in order to select the required columns.
- Execute the query and review the results.

► Task 2: Create a correlated subquery with comparison operators

- Create a correlated subquery that returns a list of sales where the quantity of sales is less than the average quantity of sales for that product.
- Use the **ProductID** and **OrderQty** columns on the **Sales.SalesOrderDetail** table.
- Execute the query and review the results.

Results: After this exercise, you should have created a correlated subquery and also created a correlated subquery with comparison operators.

Exercise 3: Comparing Subqueries with Joins and Temporary Tables

Scenario

You have also been asked by the Senior Database Administrator to prepare several reports with system performance in mind:

- List that contains prices for chainring bolts using a join instead of a subquery.
- Test report using temporary tables.

The main tasks for this exercise are as follows:

1. Create a Subquery and a Join that Produce the same Result Set.
2. Create a Temporary Table.

► Task 1: Create a subquery and a join that produce the same result set

- Create a subquery to select the name of any products that have a list price equal to the list price of 'Chainring Bolts' on the **Production.Product** table.
- Create and execute a join to select the name of any products that have a list price equal to the list price of 'Chainring Bolts' on the **Production.Product** table.

► Task 2: Create a temporary table

- Create a query that creates a temporary table with a primary key value of 1.
- In the same query, select all columns from the temporary table.
- Execute the query and review the results.

Results: After this exercise you should have created a subquery and a join that produced the same result set. You should have also created a temporary table.

Exercise 4: Using Common Table Expressions

Scenario

Finally, you have been asked by the Director of Sales to prepare the following reports:

- List of the total number of sales orders and most recent sales order date for each salesperson.
- Hierarchical list of employees including manager ID, employee ID, and employee title limited to the Research and Development Group.

The main tasks for this exercise are as follows:

1. Create a Common Table Expression.
2. Create a Recursive Query using a Common Table Expression.

► Task 1: Create a Common Table Expression (CTE)

- Create a CTE named **Sales_CTE** that returns a list of the total number of sales orders and most recent sales order date for each salesperson.
- Refer to the sample file **E:\MOD05\Labfiles\Solution\Sales_CTE.sql** as needed.

► Task 2: Create a recursive query using a CTE

- Create a recursive query using a CTE that returns a hierarchical list of product assemblies and components required to build the bicycle for **ProductAssemblyID = 800**. Use the **AssemblyID**, **PerAssemblyQty**, and **EndDate** columns of the **Production.BillOfMaterials** table. Also, create a column called **ComponentLevel** that has a value of 0 for the anchor member, and increases by 1 for each level of the hierarchy.
- Refer to the sample file **E:\MOD05\Labfiles\Solution\ProductBOM.sql** as needed.

Results: After this exercise, you should have created a common table expression. You should have also created a recursive query using a common table expression.

Lab Shutdown

After you complete the lab, you must shut down the **2778A-NY-SQL-01** virtual machine and discard any changes.

Module Review and Takeaways

- Review Questions
- Best Practices

Review Questions

1. A subquery nested in the outer SELECT statement has what components?
2. What clauses can the SELECT query of a subquery include? What clauses can it not include?
3. What are the three basic types of subqueries?

Best Practices related to Subqueries with EXISTS

Supplement or modify the following best practices for your own work situations:

- Subqueries that are introduced with EXISTS are a bit different from other subqueries in the following ways:
 - The keyword EXISTS is not preceded by a column name, constant, or other expression.
 - The select list of a subquery introduced by EXISTS almost always consists of an asterisk (*). There is no reason to list column names because you are just testing whether rows that meet the conditions specified in the subquery exist.
- The EXISTS keyword is important because frequently there is no alternative, nonsubquery formulation. Although some queries that are created with EXISTS cannot be expressed any other way, many queries can use IN or a comparison operator modified by ANY or ALL to achieve similar results.

Module 6

Modifying Data in Tables

Contents:

Lesson 1: Inserting Data into Tables	6-3
Lesson 2: Deleting Data from Tables	6-10
Lesson 3: Update Data in Tables	6-17
Lesson 4: Overview of Transactions	6-23
Lab: Modifying Data	6-33

Module Overview

- Inserting Data into Tables
- Deleting Data from Tables
- Updating Data in Tables
- Overview of Transactions

You can use Transact-SQL in Microsoft® SQL Server® 2008 to add, remove, or update data in tables using several different T-SQL statements. In this module, you will learn about inserting data into tables, deleting data from tables, and updating data in tables. You will also learn the basics of a transaction, which is a sequence of operations such as data insertions, deletions, or updates, that are performed as a single logical unit of work.

Lesson 1

Inserting Data into Tables

- **INSERT Fundamentals**
- **INSERT Statement Definitions**
- **INSERT Statement Examples**
- **Inserting Values into Identity Columns**
- **INSERT and the OUTPUT Clause**

INSERT appends new rows to a table. In this lesson, you will learn about inserting rows into a table, differentiating types of INSERT statements, and inserting values into identity columns. You will also learn about using the OUTPUT clause with the INPUT statement.

INSERT Fundamentals

- The **INSERT** statement adds one or more new rows to a table
- **INSERT** inserts *data_values* as one or more rows into the specified *table_or_view*
- *column_list* is a list of column names used to specify the columns for which data is supplied

INSERT Syntax:

```
INSERT [INTO] table_or_view [(column_list)] data_values
```

Key Points

The INSERT statement inserts *data_values* as one or more rows into the specified table or view.

- INSERT and SELECT statements can be used to add rows to a table in the following ways:
 - Use the INSERT statement to specify values either explicitly or from a subquery.
 - Use the SELECT statement with the INTO clause.
- The INSERT statement inserts *data_values* as one or more rows into the specified table or view.
- *column_list* is a list of column names, separated by commas that can be used to specify the columns for which data is supplied. If *column_list* is not specified, all the columns in the table or view receive data.

INSERT Statement Definitions

INSERT using SELECT

```
INSERT INTO MyTable (PriKey, Description)
SELECT ForeignKey, Description
FROM SomeView
```

INSERT using EXECUTE

```
CREATE PROCEDURE dbo.SomeProcedure
INSERT dbo.SomeTable
EXECUTE SomeProcedure
```

INSERT using TOP

```
INSERT TOP (#) INTO SomeTableA
SELECT SomeColumnX, SomeColumnY
FROM SomeTableB
```



Key Points

INSERT appends new rows to a table.

- Using a SELECT subquery lets more than one row be inserted at the same time. The select list of the subquery must match the column list of the INSERT statement.
- Generally speaking, you can use INSERT with EXECUTE to retrieve some set of data using a stored procedure and then store that result set into a new table.
- You would use the TOP clause in an INSERT statement along with a subquery to add the top N (some number) rows from one table or join into another table.

INSERT Statement Examples

Using a Simple INSERT Statement

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

Inserting Multiple Rows of Data

```
INSERT INTO Production.UnitMeasure  
VALUES (N'F2', N'Square Feet', GETDATE()),  
(N'Y2', N'Square Yards', GETDATE());
```



Key Points

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.
- Inserting more than one row of values requires the VALUES list to be in the same order as the columns in the table, to have a value for each column in the table, or for the column_list to explicitly specify the column that stores each incoming value. The maximum number of rows that can be inserted in a single INSERT statement is 1000. To insert more than 1000 rows, create multiple INSERT statements.

Inserting Values into Identity Columns

- **column_list and VALUES must be used to insert values into an identity column, and the SET IDENTITY_INSERT option must be ON for the table**

```
CREATE TABLE dbo.T1 (column_1 int  
IDENTITY, column_2 VARCHAR(30));  
GO  
INSERT T1 VALUES ('Row #1');  
INSERT T1 (column_2) VALUES ('Row #2');  
GO  
SET IDENTITY_INSERT T1 ON;  
GO  
INSERT INTO T1 (column_1,column_2)  
VALUES (-99, 'Explicit identity  
value');  
GO  
SELECT column_1, column_2  
FROM T1;
```

Key Points

If an identity column exists for a table with frequent deletions, gaps can occur between identity values. To fill an existing gap, you can explicitly identify identity columns in an INSERT statement.

- The SET IDENTITY_INSERT ON statement overrides the IDENTITY property for the column.
- The VALUES keyword specifies the values for one or more appended rows of a table. In the case of an IDENTITY column, it inserts a new value known as a seed that SQL Server uses to automatically generate sequential numbers for the rest of the IDENTITY rows in the table.

INSERT and the OUTPUT Clause

- **Using OUTPUT in an INSERT statement returns information from each row affected by the INSERT statement**

Syntax:

```
INSERT SomeTable  
OUTPUT dml_select_list INTO  
(@table_variable | output_table) (column_list)
```

Example Query:

```
DECLARE @MyTableVar table( ScrapReasonID smallint,  
                           Name varchar(50),  
                           ModifiedDate datetime);  
  
INSERT Production.ScrapReason  
      OUTPUT INSERTED.ScrapReasonID, INSERTED.Name,  
            INSERTED.ModifiedDate  
      INTO @MyTableVar  
      VALUES (N'Operator error', GETDATE());
```



Key Points

The OUTPUT clause may be useful to retrieve the value of identity or computed columns after an INSERT operation.

- The following restrictions apply to the target of the outer INSERT statement:
 - The target cannot be a remote table, view, or common table expression.
 - The target cannot have a FOREIGN KEY constraint, or be referenced by a FOREIGN KEY constraint.
- Results can be returned to the processing application for use in such things as confirmation messages, archiving, and other such application requirements.

Demonstration: Inserting Data into Tables

In this demonstration, you will see how to:

- Insert a Single Row into a Table
- Insert Multiple Rows into a Table
- Insert Values into Identity Columns
- Use the OUTPUT Clause with the INSERT Statement

Question: Why is the VALUES clause used when appending data into a table?

Question: Under what circumstances must the *column_list* portion of the INSERT statement be defined?

Lesson 2

Deleting Data from Tables

- **DELETE Fundamentals**
- **DELETE Statement Definitions**
- **Defining and Using the TRUNCATE Statement**
- **TRUNCATE versus DELETE**
- **DELETE and the OUTPUT Clause**

The **DELETE** statement removes one or more rows in a table or view. In this lesson, you will learn about deleting rows from a table. You will also learn about truncating a table and the differences between the **TRUNCATE** and **DELETE** statements. Finally, you will learn about deleting rows based on other tables and about using the **OUTPUT** clause with the **DELETE** statement.

DELETE Fundamentals

- The DELETE statement removes one or more rows in a table or view
- DELETE removes rows from the *table_or_view* parameter that meet the *search condition*
- *table_sources* can be used to specify additional tables or views that can be used by the WHERE clause

DELETE Syntax:

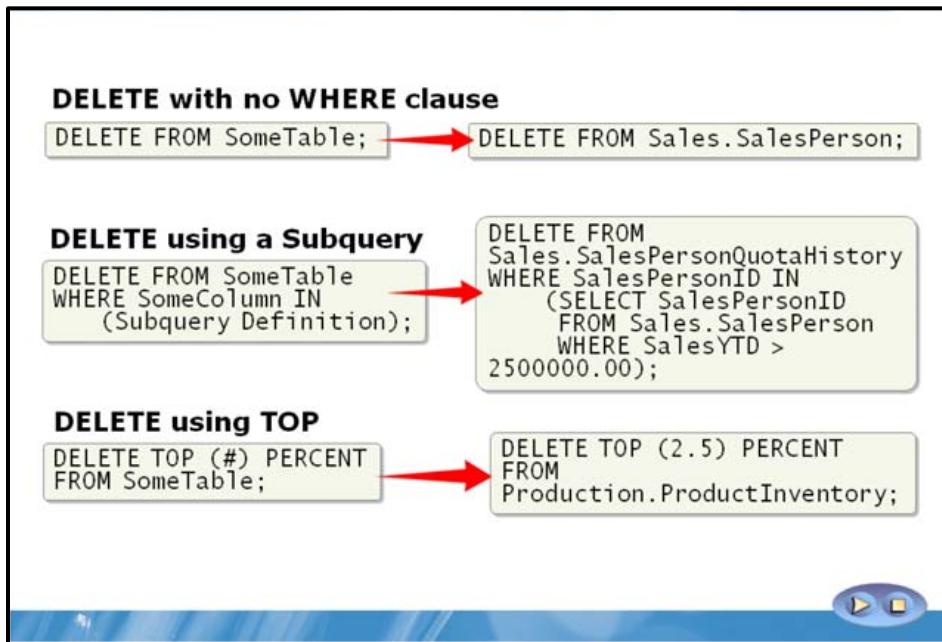
```
DELETE table_or_view  
FROM table_sources  
WHERE search_condition
```

Key Points

The DELETE statement removes one or more rows in a table or view based on the following parameters and rules.

- The parameter *table_or_view* names a table or view from which the rows are to be deleted.
- All rows in *table_or_view* that meet the qualifications of the WHERE search condition are deleted.
- If a WHERE clause is not specified, all the rows in *table_or_view* are deleted.
- Rows are not deleted from the tables named in the FROM clause, only from the table named in *table_or_view*.
- Any table that has all rows removed remains in the database, and must be removed by using the DROP TABLE statement.

DELETE Statement Definitions



Key Points

There are many ways you can remove rows from a table or view using the `DELETE` statement.

- `DELETE` can be used without a `WHERE` clause to delete all rows of a table without limitation. The `WHERE` clause of the `DELETE` statement can be defined as a subquery in order to delete rows from a base table depending on data stored in another table.
- `DELETE` can be modified with a `TOP` clause, much like `INSERT` can, in order to remove some number or percentage of rows from a table.

Defining and Using the TRUNCATE Statement

TRUNCATE TABLE Syntax

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

TRUNCATE TABLE Example

```
TRUNCATE TABLE HumanResources.JobCandidate;
```

 You cannot use TRUNCATE TABLE on tables that are referenced by a FOREIGN KEY constraint

◀ ▶ ⟲ ⟳

Key Points

TRUNCATE TABLE removes all rows from a table without logging the individual row deletions.

- The parameter *database_name* is the name of the database that the table to be truncated belongs to, *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.
- You cannot use TRUNCATE TABLE on tables that:
 - Are referenced by a FOREIGN KEY constraint, although you can truncate a table that has a foreign key that references itself.
 - Participate in an indexed view.
 - Are published by using transactional replication or merge replication.
 - For tables with one or more of these characteristics, use the DELETE statement instead.

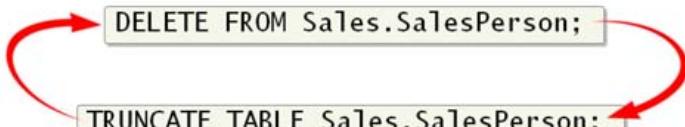
TRUNCATE versus DELETE

TRUNCATE TABLE has the following advantages over DELETE:

- Less transaction log space is used
- Fewer locks are typically used
- Zero pages are left in the table

`DELETE FROM Sales.SalesPerson;`

`TRUNCATE TABLE Sales.SalesPerson;`



Key Points

TRUNCATE TABLE is similar to the DELETE statement when no WHERE clause has been specified; however, TRUNCATE TABLE is faster and uses fewer system and transaction log resources.

- The DELETE statement removes rows one at a time and records an entry in the transaction log for each deleted row, while TRUNCATE TABLE removes the data by deallocating the data pages used to store the table data and records only the page deallocations in the transaction log.
- When the DELETE statement is executed using a row lock, each row in the table is locked for deletion, while TRUNCATE TABLE always locks the table and page but not each row.
- After a DELETE statement is executed, the table can still contain empty pages. For indexes, the delete operation can leave empty pages behind.
- TRUNCATE TABLE removes all rows from a table, but the table structure and its columns, constraints, indexes, and so on remain. To remove the table definition in addition to its data, use the DROP TABLE statement.

DELETE and the OUTPUT Clause

• Using **OUTPUT** in a **DELETE** statement removes a row from a table and returns the deleted values to a result set

Syntax:

```
DELETE SomeTable  
    OUTPUT column_list
```

Example Query:

```
DELETE Production.Culture  
    OUTPUT DELETED.*;
```

Culture

CultureID	Name	ModifiedDate
Ar	Arabic	1998-06-01
En	English	1998-06-01



Key Points

Using the OUTPUT clause with the DELETE statement returns deleted rows, or expressions based on them, as part of the DELETE operation.

- DELETE statements that use the OUTPUT clause return results that can be used in such things as confirmation messages, archiving, and other such application requirements.
- Results can also be inserted into a table or table variable.
- The clause OUTPUT DELETED.* specifies that the results of the DELETE statement, that is all columns in the deleted rows, be returned to the calling application.

Demonstration: Deleting Data from Tables

In this demonstration, you will see how to:

- Delete Rows from a Table
- Truncate a Table
- Delete Rows Based on Other Tables
- Use the OUTPUT Clause with the DELETE Statement

Question: Why should you use the TRUNCATE statement?

Question: How can you completely remove a table from a database?

Lesson 3

Updating Data in Tables

- UPDATE Fundamentals
- UPDATE Statement Definitions
- Updating with Information from another Table
- UPDATE and the OUTPUT Clause

The UPDATE statement changes existing data in a table. In this lesson, you will learn about updating rows in a table and updating rows based on other tables. You will also learn about using the OUTPUT clause with the UPDATE statement.

UPDATE Fundamentals

- **The UPDATE statement changes data values in one, many, or all rows of a table**
- **An UPDATE statement referencing a table or view can change the data in only one base table at a time**
- **UPDATE has three major clauses:**
 - **SET – comma-separated list of columns to be updated**
 - **FROM – supplies values for the SET clause**
 - **WHERE – specifies a search condition for the SET clause**

UPDATE Syntax:

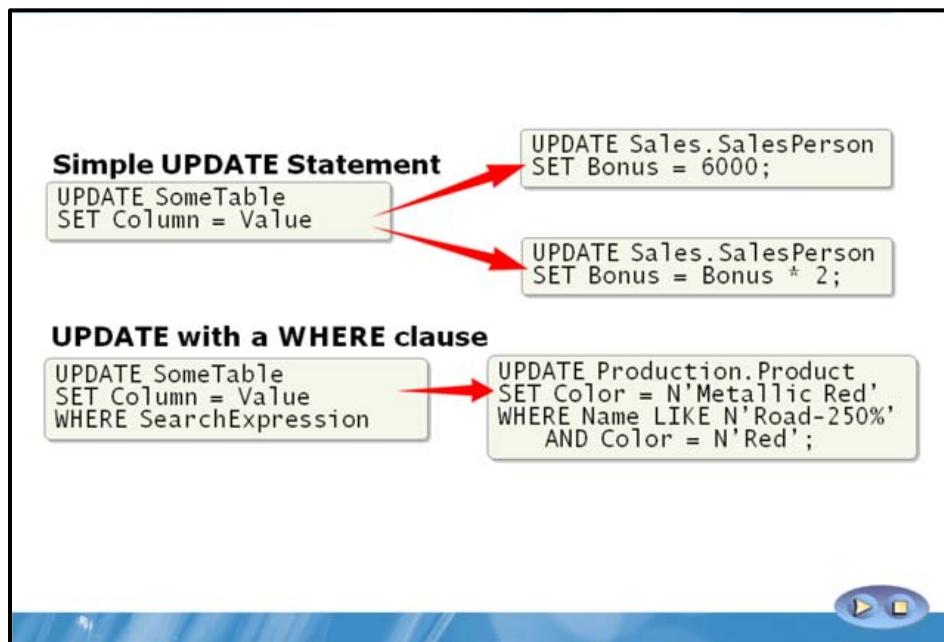
```
UPDATE table_or_view  
SET column_name = expression  
FROM table_sources  
WHERE search_condition
```

Key Points

The UPDATE statement can change data values in single rows, groups of rows, or all the rows in a table or view.

- The UPDATE statement has the following major clauses:
 - **SET**
Contains a comma-separated list of the columns to be updated and the new value for each column, in the form *column_name* = *expression*. The value supplied by the expressions includes items such as constants, values selected from a column in another table or view, or values calculated by a complex expression.
 - **FROM**
Identifies the tables or views that supply the values for the expressions in the SET clause, and optional join conditions between the source tables or views.
 - **WHERE**
Specifies the search condition that defines the rows from the source tables and views that qualify to provide values to the expressions in the SET clause.
- An UPDATE statement referencing a table or view can change the data in only one base table at a time.

UPDATE Statement Definitions



Key Points

UPDATE statements can be used in several different ways.

- UPDATE statements can be simple, short, and do not require the use of the FROM or WHERE clauses to function properly.
- You can use computed values in an UPDATE statement.
- You can use the WHERE clause to limit the rows that are updated with an UPDATE statement.

Updating with Information from Another Table

UPDATE using a Subquery

```
UPDATE SomeTable  
SET Column = Value  
FROM SomeSubquery
```

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

Before

SalesYTD
677558.4653
4557045.0459

After

SalesYTD
721382.488
4593234.5123

▶ ◀

Key Points

You can use the UPDATE statement in a subquery to update rows in one table with information from another table.

- You can use a subquery in the FROM clause of an UPDATE statement in the place of an explicit table source that acts as criteria for the update operation.
- Updating with information from another table can be useful for example when you have several tables that relate to one another even if only in business terms, such as if a product price is listed in a table named `ProductPrices` but also relates to a `ProductSales` table in some way.

UPDATE and the OUTPUT Clause

• Using **OUTPUT** in an **UPDATE** statement returns information from each row affected by the **UPDATE** statement

Syntax:

```
UPDATE SomeTable  
OUTPUT dml_select_list FROM table_source  
WHERE search_condition
```

```
DECLARE @NewTableVar table ( Dollars money );  
UPDATE Sales.SalesPerson  
SET Bonus = 10000  
OUTPUT INSERTED.Bonus INTO @NewTableVar;  
  
SELECT Dollars  
FROM @NewTableVar;
```

Dollars

Dollars

10000.00
10000.00
...
(17 row(s) affected)



Key Points

You can use the **OUTPUT** clause with the **UPDATE** statement to return updated rows as part of the **UPDATE** operation.

- In the case of an **UPDATE** statement, *dml_select_list* values are generally **INSERTED**.*SomeColumnName* or **DELETED**.*SomeColumnName*.
- The **@table_variable** argument specifies a table variable that the returned rows are inserted into instead of being returned to the caller.
- **@table_variable** must be declared before the **UPDATE** statement itself.
- Using the **OUTPUT** clause with an **UPDATE** statement can be useful for returning results from one **UPDATE** statement into a table variable that can then be used in other queries or in applications that can access SQL Server data.

Demonstration: Updating Data in Tables

In this demonstration, you will see how to:

- Update Rows in a Table
- Update Rows Based on Other Tables
- Use the OUTPUT Clause with the UPDATE Statement

Question: What are the major clauses of the UPDATE statement and what do they do?

Question: How would you write an UPDATE statement to increase the price of all products on a table by 10 percent?

Lesson 4

Overview of Transactions

- Transaction Fundamentals
- Transactions and the Database Engine
- Basic Transaction Statement Definitions
- What are Transaction Isolation Levels?
- Using Nested Transactions

As mentioned in the module introduction, a transaction is a sequence of operations performed as a single logical unit of work. In this lesson, you will learn about transactions, concepts related to transaction, managing transactions, and using nested transactions. You will also learn about transaction isolation levels and how to manage them.

Transaction Fundamentals

A Transaction:

- Is a sequence of operations performed as a single logical unit of work

• Exhibits the four ACID Properties

- Atomicity - must be an atomic unit of work
- Consistency - must leave all data in a consistent state
- Isolation - must be isolated from the modifications made by any other concurrent transactions
- Durability - persists even after system failure



Key Points

A transaction is a sequence of operations performed as a single logical unit of work.

- A logical unit of work must exhibit the four 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.

- Isolation

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.

- 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 programmers are responsible for starting and ending transactions at points that enforce the logical consistency of the data.

Transaction and the Database Engine

The Database Engine provides:

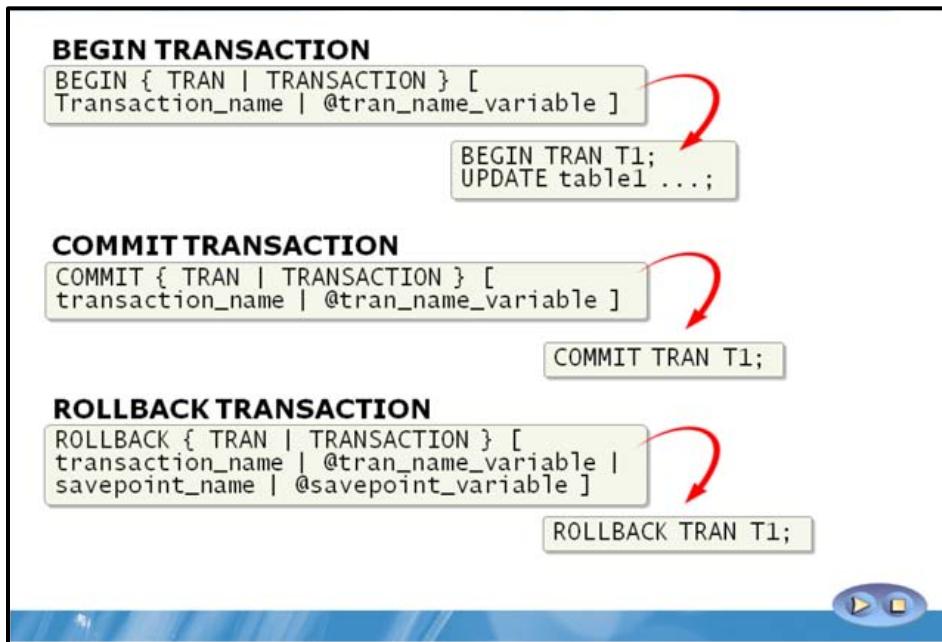
- **Locking facilities that preserve transaction isolation**
 - Transaction Isolation Levels control when locks are taken and how long they are held
- **Logging facilities that ensure transaction durability**
 - Write-ahead log (WAL) guarantees no data modifications are written before they are logged
 - Checkpoints write records to a data file and contain lists of all active transactions
- **Transaction management features that enforce transaction atomicity and consistency**
 - Transactions must be successfully completed or their modifications are undone

Key Points

The Database Engine provides locking facilities to preserve transaction isolation, logging facilities that ensure transaction durability, and transaction management features that enforce atomicity and consistency.

- Transactions specify an isolation level that defines the degree to which one transaction must be isolated from resource or data modifications made by other transactions.
- SQL Server uses a write-ahead log (WAL), which guarantees that no data modifications are written to disk before the associated log record is written to disk.
- Checkpoints flush dirty data pages from the buffer cache of the current database to disk. This minimizes the active portion of the log that must be processed during a full recovery of a database.

Basic Transaction Statement Definitions



Key Points

BEGIN TRANSACTION marks the starting point of an explicit, local transaction, and COMMIT TRANSACTION marks the end of a successful transaction.

- BEGIN TRANSACTION represents a point at which the data referenced by a connection is logically and physically consistent. If errors are encountered, all data modifications made after the BEGIN TRANSACTION can be rolled back to return the data to this known state of consistency.
- Each transaction lasts until either it completes without errors and COMMIT TRANSACTION is issued to make the modifications a permanent part of the database, or errors are encountered and all modifications are erased with a ROLLBACK TRANSACTION statement.
- If a run-time statement error (such as a constraint violation) occurs in a batch, the default behavior in the Database Engine is to roll back only the statement that generated the error.

Demonstration: Creating a Transaction

In this demonstration, you will see how to:

- Create and Commit a New Transaction

Question: How can a transaction be rolled back and when would you want to roll a transaction back?

What are Transaction Isolation Levels?

- **Transaction Isolation Levels control**
 - Whether locks are taken when data is read
 - How long read locks are held
 - How a read operation referencing rows acts
- **Choosing a transaction isolation level does not affect the locks acquired to protect data modifications**
- **The levels are READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SNAPSHOT, and SERIALIZABLE**

Syntax

```
SET TRANSACTION ISOLATION LEVEL <level>;
```

Key Points

As mentioned previously, transactions specify an isolation level that defines the degree to which one transaction must be isolated from modifications made by other transactions.

- Transaction isolation levels control:
 - Whether locks are taken when data is read, and what type of locks are requested.
 - How long the read locks are held.
 - Whether a read operation referencing rows can be modified by another transaction:
 - Blocks until the exclusive lock on the row is freed.
 - Retrieves the committed version of the row that existed at the time the statement or transaction started.
 - Reads the uncommitted data modification.
- Choosing a transaction isolation level does not affect the locks acquired to protect data modifications.
- A transaction always gets an exclusive lock on any data it modifies, and holds that lock until the transaction completes, regardless of the isolation level set for that transaction.
- The transaction isolation levels available are as follows:
 - READ UNCOMMITTED
 - Specifies that statements can read rows that have been modified by other transactions but not yet committed.
 - READ COMMITTED
 - Specifies that statements cannot read data that has been modified but not committed by other transactions. This prevents dirty reads.

- REPEATABLE READ

Specifies that statements cannot read data that has been modified but not yet committed by other transactions and that no other transactions can modify data that has been read by the current transaction until the current transaction completes.

- SNAPSHOT

Specifies that data read by any statement in a transaction will be the transactionally consistent version of the data that existed at the start of the transaction.

- SERIALIZABLE

Locks all tables in all SELECT statements in a transaction.

Demonstration: Setting Transaction Isolation Levels

In this demonstration, you will see how to:

- Set a Transaction Isolation Level

Question: How long does SQL Server hold a shared lock created by the SET TRANSACTION ISOLATION LEVEL statement?

Using Nested Transactions

• Explicit transactions can be nested to support transactions in stored procedures

```
CREATE TABLE TestTrans(Cola INT PRIMARY KEY,
                      Colb CHAR(3) NOT NULL);
GO
CREATE PROCEDURE TransProc @PriKey INT, @CharCol1 CHAR(3) AS
BEGIN TRANSACTION InProc
INSERT INTO TestTrans VALUES (@PriKey, @CharCol1)
INSERT INTO TestTrans VALUES (@PriKey + 1, @CharCol1)
COMMIT TRANSACTION InProc;
GO
BEGIN TRANSACTION OutOfProc; /* Starts a transaction */
GO
EXEC TransProc 1, 'aaa';
GO
ROLLBACK TRANSACTION OutOfProc; /* Rolls back the outer
transaction */
GO
EXECUTE TransProc 3, 'bbb';
GO
SELECT * FROM TestTrans;
```

Cola Colb

1	bb
2	bb



Key Points

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.
- Committing inner transactions is ignored by the SQL Server Database Engine. The transaction is either committed or rolled back based on the action taken at the end of the outermost transaction.

Demonstration: Using Nested Transactions

In this demonstration, you will see how to:

- Create a Nested Transaction

Question: What did the SELECT statement indicate to you?

Lab: Modifying Data

- Exercise 1: Inserting Data into Tables
- Exercise 2: Deleting Data from Tables
- Exercise 3: Updating Data in Tables
- Exercise 4: Working with Transactions

Logon information

Virtual machine	NY-SQL-01
User name	Administrator
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Exercise 1: Inserting Data into Tables

Scenario

You are the database developer at Adventure Works. You have been asked by the senior database administrator to:

- Include new product information about an existing product:

UnitMeasureCode	Name	ModifiedDate
F2	Square Feet	GETDATE()

- Include information about new departments and their managers:

BusinessEntityID	PhoneNumber	PhoneNumberTypeID	ModifiedDate
1705	864-555-2101	3	GETDATE()
1706	712-555-0118	1	GETDATE()

- Include information about a faulty product:

Name	ModifiedDate
Operator error	GETDATE()

The main tasks for this exercise are as follows:

- Launch **SQL Server Management Studio**.
- Create an INSERT Statement that Adds Values to a Table.
- Create an INSERT Statement Using the INTO Syntax.
- Create an INSERT Statement Using the OUTPUT Syntax.

► Task 1: Launch SQL Server Management Studio

- Start the **2778A-NY-SQL-01** virtual machine, and log on as **Administrator** with the password of **Pa\$\$w0rd**.
- Open **SQL Server Management Studio**.

► Task: 2 Create an INSERT statement that adds values to a table

- Create an INSERT statement that will include new product information about an existing product into the **Production.Unit** measure table.
- Use the information provided in the scenario above.
- Execute the query and review the results.

► Task 3: Create an INSERT statement that adds multiple rows to a table

- Create an INSERT statement that will include information about new departments and their managers into the **Person.PersonPhone** table.
- Use the information provided in the scenario above.
- Execute the query and review the results.

► **Task 4: Create an INSERT statement using the OUTPUT syntax**

- Create an INSERT statement that will include information about a faulty product into a new table created with the following query:

```
DECLARE @MyTableVar table( ScrapReasonID smallint,  
                           Name varchar(50),  
                           ModifiedDate datetime);
```

- Insert the **ScrapReasonID**, **Name**, and **ModifiedDate** columns and rows from the **Production.ScrapReason** table into **@MyTableVar** using the INSERT and OUTPUT statements.
- Display the result set of the table variable and display the result set of the table.

Results: After this exercise, you should have launched SQL Server Management Studio, created and executed an INSERT statement that adds values to a table, created and executed an INSERT statement using the INTO syntax, and created and executed an INSERT statement using the OUTPUT syntax.

Exercise 2: Deleting Data from Tables

Scenario

The sales manager has asked you to make the following changes to the database:

- Delete all instances of product cost history where standard cost is greater than \$1000.00.

The director of sales has now asked you to make the following updates to the database:

- Remove rows from the sales person quota history based on the year-to-date sales stored in the **SalesPerson** table, where year-to-date sales are greater-than \$1,000,000.00.
- Delete the **Sales.ShoppingCartItem** table and display the results of the deletion.
- Truncate the **Production.TransactionHistory** table and display the results.

The main tasks for this exercise are as follows:

1. Create a DELETE Statement Using the WHERE Syntax.
2. Create a DELETE Statement Using the FROM Syntax.
3. Create a DELETE Statement Using the OUTPUT Syntax.
4. Create a TRUNCATE TABLE Statement.

► Task 1: Create a DELETE statement using the WHERE syntax

- Create a DELETE statement that deletes all instances of product cost history where standard cost is greater than \$1000.00.
- Execute the query and review the results.

► Task 2: Create a DELETE statement using the FROM syntax

- Create a DELETE statement using the FROM syntax that removes rows from the sales person quota history based on the year-to-date sales stored in the **SalesPerson** table, where year-to-date sales are greater-than \$1,000,000.00.
- Execute the query and review the results.

► Task 3: Create a DELETE statement using the OUTPUT syntax

- Create a DELETE statement using the OUTPUT syntax that deletes the **Sales.ShoppingCartItem** table and display the results of the deletion.
- Execute the query and review the results.

► Task 4: Create a TRUNCATE TABLE statement

- Create a TRUNCATE TABLE statement that truncates the **Production.TransactionHistory** table, displays the count of table rows before the statement is executed, and also displays the count of table rows after the statement is executed.
- Execute the query and review the results.

Results: After this exercise, you should have created and executed a DELETE statement using the WHERE syntax and created and executed a DELETE statement using the FROM syntax. You should have also created and executed a DELETE statement using the OUTPUT syntax and then created and executed a TRUNCATE TABLE statement.

Exercise 3: Updating Data in Tables

Scenario

A vice president at the AdventureWorks company has now asked you to make the following updates to the database:

- Update the **Sales.SalesPerson** table to set the bonus amount to \$6000, the commission percent to .10, and the sales quota to NULL.
- Change all instances of **Road-250** products on the **Production.Product** table with the color **Red** to the color **Metallic Red**.
- Modify the year-to-date sales of the **Sales.SalesPerson** table to reflect the most recent sales recorded in the **Sales.SalesOrderHeader** table.
- Update the **VacationHours** column in the **HumanResources.Employee** table by 25 percent for the first 10 rows.

The main tasks for this exercise are as follows:

1. Create an UPDATE Statement Using the SET Syntax.
2. Create an UPDATE Statement Using the WHERE Syntax.
3. Create an UPDATE Statement Using the FROM Syntax.
4. Create an UPDATE Statement Using the OUTPUT Syntax.

► Task 1: Create an UPDATE statement using the SET syntax

- Create an UPDATE statement using the SET syntax that updates the **Sales.SalesPerson** table to set the bonus amount to \$6000, the commission percent to .10, and the sales quota to NULL.
- Execute the query and review the results.

► Task 2: Create an UPDATE statement using the WHERE syntax

- Create an UPDATE statement using the WHERE syntax that changes all instances of **Road-250** products on the **Production.Product** table with the color **Red** to the color **Metallic Red**.
- Execute the query and review the results.

► Task 3: Create an UPDATE statement using the FROM syntax

- Create an UPDATE statement using the FROM syntax that modifies the year-to-date sales of the **Sales.SalesPerson** table to reflect the most recent sales recorded in the **Sales.SalesOrderHeader** table.
- Execute the query and review the results.

► Task 4: Create an UPDATE statement using the OUTPUT syntax

- Create an UPDATE statement using the OUTPUT syntax that updates the **VacationHours** column in the **HumanResources.Employee** table by 25 percent for the first 10 rows.
- Execute the query and review the results.

Results: After this exercise, you should have created and executed an UPDATE statement using the SET syntax, created and executed an UPDATE statement using the WHERE IN (SELECT) syntax, and created and executed an UPDATE statement using the FROM syntax. You should have also created and executed an UPDATE statement using the OUTPUT syntax.

Exercise 4: Working with Transactions

Scenario

Finally, the senior database administrator has asked you to implement the following transactions on the database:

- Create and commit a transaction that will delete **JobCandidateID** 13 from the **HumanResources.JobCandidate** table.
- Create and commit a repeatable read transaction that will return all rows from the **HumanResources.EmployeePayHistory** table and all rows from the **HumanResources.Department** table.

The main tasks for this exercise are as follows:

1. Create a Simple Transaction.
2. Set a Transaction Isolation Level.

► Task 1: Create a simple transaction

- Create and commit a transaction that will delete **JobCandidateID** 13 from the **HumanResources.JobCandidate** table.
- Execute the query and review the results.

► Task 2: Set a Transaction Isolation Level

- Create and commit a repeatable read transaction that will return all rows from the **HumanResources.EmployeePayHistory** table and all rows from the **HumanResources.Department** table.
- Execute the query and review the results.

Results: After this exercise, you should have created a simple transaction. You should have also set the transaction isolation level for a new transaction.

Lab Shutdown

After you complete the lab, you must shut down the **2778A-NY-SQL-01** virtual machine and discard any changes.

Module Review and Takeaways

- Review Questions
- Best Practices

Review Questions

1. What are the four properties of a logical unit of work?
2. How can the INSERT and SELECT statements be used to add rows to a table?
3. What happens when a WHERE clause is not specified in a DELETE statement?
4. What are the major clauses of the UPDATE statement?

Best Practices related to specifying and enforcing transactions

Supplement or modify the following best practices for your own work situations:

- SQL programmers are responsible for starting and ending transactions at points that enforce the logical consistency of the data.
- The programmer must define the sequence of data modifications that leave the data in a consistent state relative to the organization's business rules.
- The programmer must include these modification statements in a single transaction so that the SQL Server Database Engine can enforce the physical integrity of the transaction.

Best Practices related to inserting rows into a table

Supplement or modify the following best practices for your own work situations:

- If a value is being loaded into columns with a **char**, **varchar**, or **varbinary** data type, the padding or truncation of trailing blanks (spaces for **char** and **varchar**, zeros for **varbinary**) is determined by the SET ANSI_PADDING setting defined for the column when the table was created.
- If an empty string (' ') is loaded into a column with a varchar or text data type, the default operation is to load a zero-length string.
- If an INSERT statement violates a constraint or rule, or if it has a value incompatible with the data type of the column, the statement fails and the Database Engine displays an error message.

Best Practices related to Deleting Rows from a Table

Supplement or modify the following best practices for your own work situations:

- DELETE can be used in the body of a user-defined function if the object modified is a **table** variable.
- The DELETE statement may fail if it tries to remove a row referenced by data in another table with a FOREIGN KEY constraint. If the DELETE removes multiple rows, and any one of the removed rows violates a constraint, the statement is canceled, an error is returned, and no rows are removed.
- If you want to delete all the rows in a table, use the DELETE statement without specifying a WHERE clause, or use TRUNCATE TABLE.
- TRUNCATE TABLE is faster than DELETE and uses fewer system and transaction log resources.



Note: If you do use the TRUNCATE statement on a table, you will reset the identity seed of any IDENTITY column in that table.

Best Practices related to Updating Rows in a Table

Supplement or modify the following best practices for your own work situations:

- UPDATE statements are allowed in the body of user-defined functions only if the table being modified is a **table** variable.
- If an update to a row violates a constraint or rule, violates the NULL setting for the column, or the new value is an incompatible data type, the statement is canceled, an error is returned, and no records are updated.
- The results of an UPDATE statement are undefined if the statement includes a FROM clause that is not specified in such a way that only one value is available for each column occurrence that is updated, that is if the UPDATE statement is not deterministic.

Module 7

Querying Metadata, XML, and Full-Text Indexes

Contents:

Lesson 1: Querying Metadata	7-3
Lesson 2: Overview of XML	7-14
Lesson 3: Querying XML Data	7-20
Lesson 4: Overview of Full-Text Indexes	7-26
Lesson 5: Querying Full-Text Indexes	7-31
Lab: Querying Metadata, XML, and Full-Text Indexes	7-38

Module Overview

- **Querying Metadata**
- **Overview of XML**
- **Querying XML Data**
- **Overview of Full-Text Indexes**
- **Querying Full-Text Indexes**

Queries in Microsoft® SQL Server® 2008 often involve data that does not reside in simple data types and techniques that go beyond ordinary SELECT statements. In this module, we will discuss what metadata is and what it can be used for within business processes, how data is organized and can be accessed in XML format, and what full-text indexes are and how they can be used.

Lesson 1

Querying Metadata

- What Is Metadata?
- Compatibility Views
- System Catalog Views
- System Catalog View Examples
- Information Schema Views
- Information Schema View Examples
- Dynamic Management Views and Functions
- Dynamic Management Views and Functions Examples
- System Stored Procedures and Functions

In this lesson, we will discuss what metadata is and the different techniques that can be used to access it.

What Is Metadata?

Definition: Metadata is Data about Data.

- Adds Context to Data
- Hides Complexity From End Users
- Used in Determination of Data Types
- Reveals the Structure & Relationships Between Data
- Determines Changes in Data and What Changes Were Made
- Used for Type Checking, Data Validation and Formatting of Data



Key Points

Metadata is descriptive information about the data or database.

- An example is a customer information database in which there will be a column for customer name. The end user will see the column's values for each row as the name of each customer.
 - The metadata for the customer name will provide information regarding the data type of the column as well as the maximum length of the column.
 - Metadata will also provide information about what other values are stored within the same table as the customer name as well as the relationship of that table with other tables in the database.
- Metadata can also consist of statistical information. In our customer name example, metadata can provide information regarding the number of times that the customer name has been queried, how many unique names exist within the database, or which names have been recently modified.

Compatibility Views

- Applications developed with previous versions of SQL Server® may use metadata views that are specific to that version
- Compatibility views are provided for backward compatibility for such applications
- Compatibility views should be used for backward compatibility only
- Compatibility views do not provide information regarding features that are new to the current version of SQL Server

```
SELECT *
FROM sys.sysobjects;
```

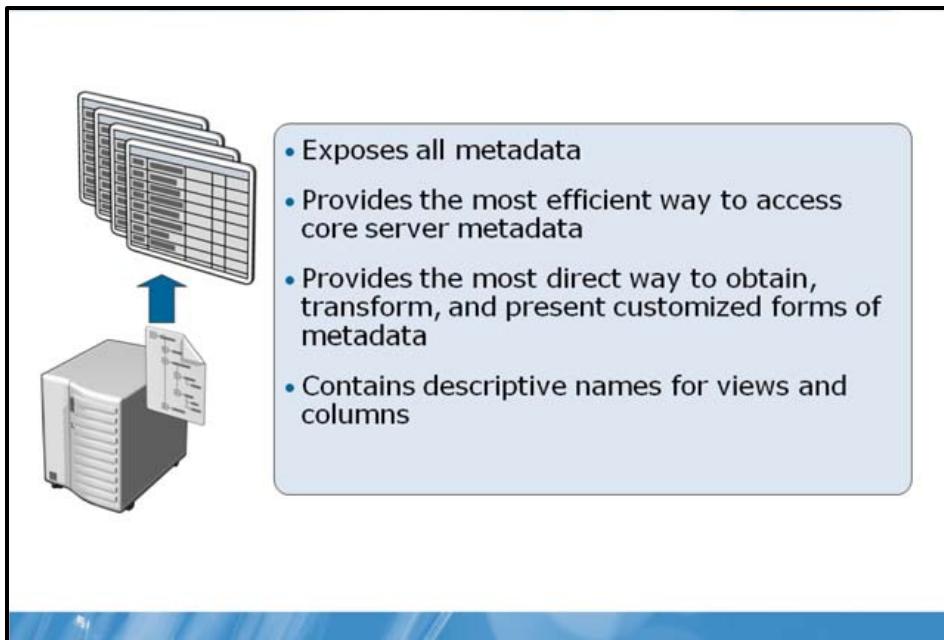
Name	Id	Xtype	Uid	Info	status
events	-414	V	4	0	0
event_notifications	-413	V	4	0	0
triggers	-412	V	4	0	0
procedures	-411	V	4	0	0
foreign_key_columns	-410	V	4	0	0

Key Points

Compatibility views are provided for applications that use metadata table references from Microsoft SQL Server 2000.

- Compatibility views should only be used for backward compatibility.
- Compatibility views do not display information regarding features implemented in SQL Server since version 2000.

System Catalog Views



Key Points

System Catalog Views provide the most direct and lowest level access to metadata in SQL Server.

- Metadata in SQL Server is stored in system tables that should never be modified directly.
- As SQL Server is enhanced and new versions are released, the structure of the system tables changes. SQL Server provides system catalog views that provide the metadata information from the system tables in a standard format.
- System Catalog Views can remain unchanged even though the underlying system tables have changed.
- Using the system catalog views are the most direct and efficient way of interacting with the metadata.

System Catalog View Examples

The slide displays three examples of System Catalog Views, each with a query script on the left and its result table on the right.

Example 1:

```
SELECT *  
FROM sys.objects;
```

name	object_id	principal_id	schema_id
sysrowsetcolumns	4	NULL	4
sysrowsets	5	NULL	4
sysallocunits	7	NULL	4
sysfiles1	8	NULL	4

Example 2:

```
SELECT name, type_desc  
FROM sys.tables;
```

Name	Type_desc
Product	USER_TABLE
Employee	USER_TABLE
Customer	USER_TABLE

Example 3:

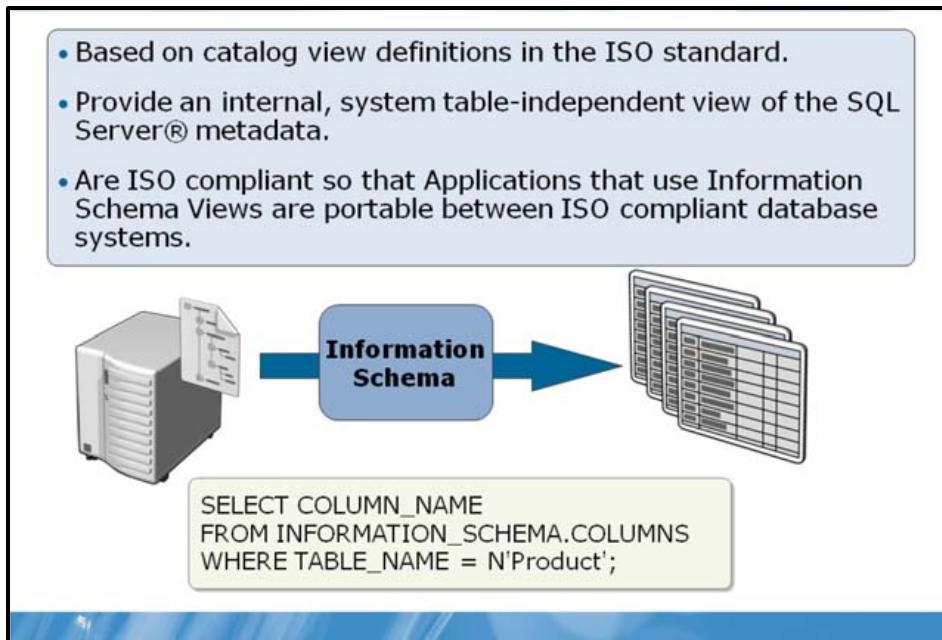
```
SELECT name, create_date,  
       modify_date  
FROM sys.views;
```

Name	Create_date	Modify_date
vwProducts	7/12/2007	7/12/2007
vwCustomer	7/14/2007	3/22/2008
vwProducts	7/12/2007	2/3/2008
vwOrders	7/12/2007	7/12/2007

Key Points

- System Catalog Views provide simple, direct access to metadata.
- System Catalog Views are available for the most common types of objects in SQL Server as well as their attributes.

Information Schema Views



Key Points

The Information Schema provides the same information as the system catalog views but also provides a layer of abstraction making the metadata more logical and easier to work with.

- This layer of abstraction also provides protection from underlying system table changes.
- The Information Schema provides a layer of abstraction to the system catalog.
- Information schema views provide an internal, system table-independent view of the SQL Server metadata.
- Information schema views enable applications to work correctly although significant changes have been made to the underlying system tables.
- Information schema views are based on the ISO standard and applications that use them are portable to other database systems that are also ISO compliant.

Information Schema View Examples

The screenshot shows three examples of Information Schema Views in SQL Server Management Studio:

- Example 1:** A query to select column names from the INFORMATION_SCHEMA.COLUMNS view for the 'Product' table.

```
SELECT COLUMN_NAME  
FROM INFORMATION_SCHEMA.COLUMNS  
WHERE TABLE_NAME = N'Product';
```

Result set (COLUMN_NAME):

COLUMN_NAME
Name
ProductNumber
MakeFlag
FinishedGoodsFlag
- Example 2:** A query to select table names from the INFORMATION_SCHEMA.VIEWS view for the 'Sales' schema.

```
SELECT TABLE_NAME  
FROM INFORMATION_SCHEMA.VIEWS  
WHERE TABLE_SCHEMA = N'Sales';
```

Result set (TABLE_NAME):

TABLE_NAME
vIndividualCustomer
vPersonDemographics
vSalesPerson
vStoreWithContacts
- Example 3:** A query to select table names from the INFORMATION_SCHEMA.TABLES view for the 'Person' schema.

```
SELECT TABLE_NAME  
FROM INFORMATION_SCHEMA.TABLES  
WHERE TABLE_SCHEMA = N'Person';
```

Result set (TABLE_NAME):

TABLE_NAME
AddressType
StateProvince
BusinessEntity
ContactType

Key Points

- The Information Schema provides metadata without being dependent on the system catalog structure.
- Information Schema Views are available for the most common types of objects in SQL Server as well as their attributes.

Dynamic Management Views and Functions

- Act as a mechanism to allow you to look at the internal workings of SQL Server using TSQL
- Provide an easy method for monitoring the internal state and health of SQL Server
- Provide information for a wide variety of categories

Category	Examples	
	View	Function
Execution	dm_exec_requests	dm_exec_sql_text
Index	dm_db_index_usage_stats	dm_db_missing_index_columns
I/O	dm_io_pending_io_requests	dm_io_virtual_file_stats
Operating System	dm_os_sys_info	

Key Points

- Dynamic Management Views (DMVs) and Dynamic Management Functions (DMFs) are mechanisms that provide activity information about SQL Server.
- This information is available for a wide variety of categories including the database engine activities (execution, indexing, etc.), replication, Service Broker, and so on.
- Dynamic Management Views and Functions provide information regarding the internal state and health of SQL Server.

Dynamic Management Views and Functions Examples

The screenshot shows three examples of Dynamic Management Views and Functions:

- Example 1:** A query to count the number of requests by command. The results are shown in a table:

Cnt	Command
1	BRKR EVENT HNDLR
3	BRKR TASK
1	CHECKPOINT
5	FSAGENT TASK

- Example 2:** A query to get database statistics for AdventureWorks. The results are shown in a table:

database_id	num_of_reads	num_of_writes
7	8	7

- Example 3:** A query to get system information. The results are shown in a table:

cpu_count	physical_memory_in_bytes	sqlserver_start_time
1	960995328	2008-08-14 12:41:30.423

Key Points

Dynamic Management Views provide activity information about SQL Server. Some examples include:

- The sys.dm_exec_requests view provides information regarding process requests that have occurred in the current database.
- The sys.dm_io_virtual_file_stats function provides information regarding the underlying file system of the current database.
- The sys.dm_os_sys_info view provides information regarding the hardware and operating system supporting the current SQL Server instance.

System Stored Procedures and Functions

- Provide a simple way of performing complex queries via T-SQL.

Common System Stored Procedures

- sp_databases
- sp_tables
- sp_columns
- sp_statistics
- sp_pkeys
- sp_fkeys

Common System Functions

- DATABASEPROPERTY
- COLUMNPROPERTY
- OBJECTPROPERTY
- COL_LENGTH
- DB_NAME
- DB_ID

```
SELECT COLUMNPROPERTY( OBJECT_ID('Person.Contact'),  
                      'LastName',  
                      'PRECISION') AS 'Column Length';
```

```
EXEC sp_columns @table_name = N'Department',  
                  @table_owner = N'HumanResources';
```

Key Points

System Stored Procedures and Functions are ways to get metadata information without in-depth understanding of the underlying structures.

- The procedures and functions accept parameters that can perform searches and filtering without requiring knowledge of the underlying metadata structures.

Demonstration: Querying Metadata

- Querying Metadata Using System Catalog Views
- Querying Metadata Using the Information Schema
- Querying Metadata Using Dynamic Management Views
- Querying Metadata Using System Stored Procedures and Functions

Question: Why would you choose to use the INFORMATION_SCHEMA rather than System Catalog views to obtain metadata?

Question: Why would you choose to use the System Stored Procedures and Functions to query metadata?

Lesson 2

Overview of XML

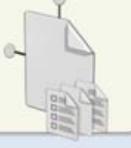
- What Is XML?
- Technical Scenarios Where XML Is Used
- Business Scenarios Where XML Is Used
- How SQL Server® 2008 Implements XML
- The XML Data Type

In this lesson, you will learn what XML is, how it can be used, and how SQL Server 2008 implements support for it.

What Is XML?

Definition: The Extensible Markup Language (XML) is an open standard recommended by W3C for creating custom markup languages.

```
<people>
  <person>
    <first_name>Robert</first_name>
    <last_name>Frost</last_name>
  </person>
  <person>
    <first_name>Frank</first_name>
    <last_name>Baker</last_name>
  </person>
</people>
```



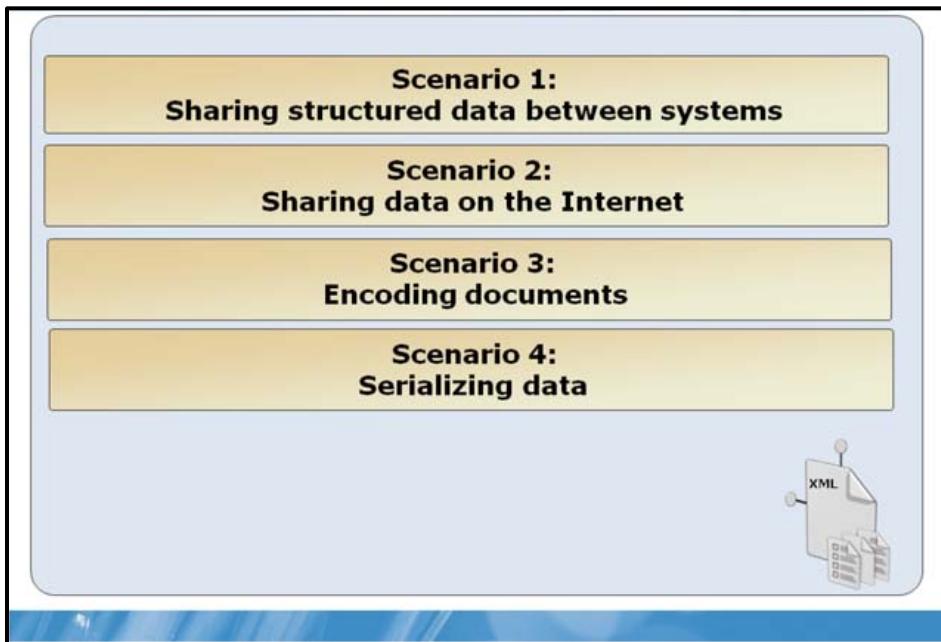
- Tag based.
- Tags are self descriptive.
- Designers are free to create their own XML structures and tags to accommodate their data.
- The XML standard does not define the structures or the content but only the format.

Key Points

XML is a general-purpose specification for creating custom markup languages.

- There is no fixed set of markup tags as with HTML.
- In XML, authors are able to define their own tags.
- The XML standard does not define the structures or the content of XML but only the format.
- The XML standard is an open standard from the W3C.

Technical Scenarios Where XML Is Used



Key Points

In today's business world, information systems are diverse and distributed. In many cases, a system of one type will need data contained in a system of another type. XML is system independent and can be used to share data between those systems.

- XML is a text-based format so data in XML format can easily be transmitted over the internet. These XML data transmissions can be secured by the same methods as are used for securing HTML transmissions.
- Information is stored in a wide variety of formats from different systems and applications. By storing the information from one application in an open XML format, that information can be utilized within other applications and systems.
- Many times, data is stored by systems and applications in a proprietary binary format. When this data needs to be transmitted by a text based protocol or stored in a text based system, it can be serialized into XML format.

Business Scenarios Where XML Is Used

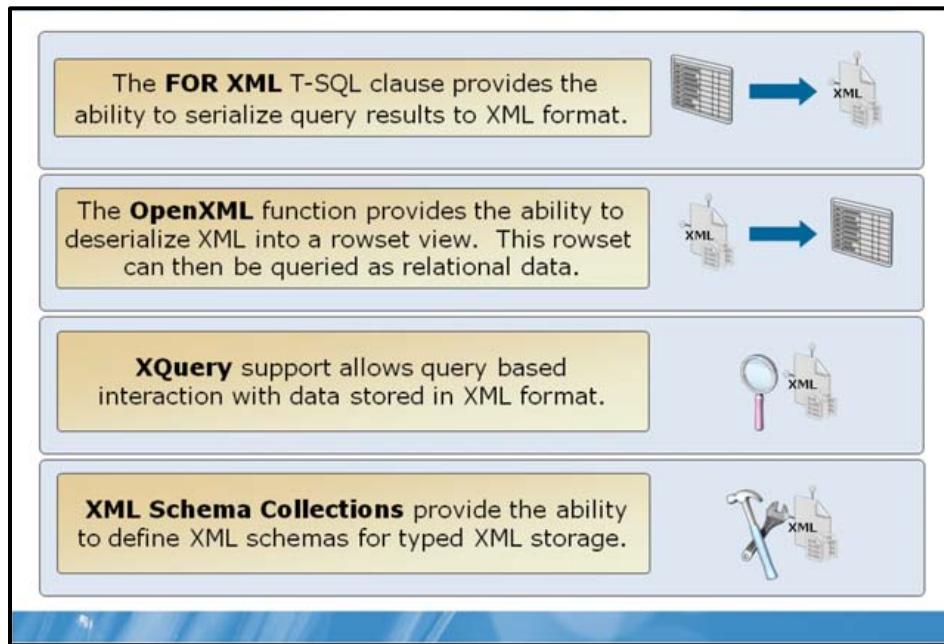


Key Points

The practical applications for XML are widespread. Some examples are:

- An auto insurance company providing services on the Internet stores specific information related to claims in a specified XML format. Exact copies of these XML documents must be maintained in the system for legal purposes.
- An automobile manufacturer procures the parts required for the company and processes invoices in XML format.
- The automobile manufacturer exposes payment information to part suppliers over the Web so that the suppliers can perform tasks automatically. This scenario demonstrates the use of XML for business integration.
- A company wants to build a content management system to help it deliver high quality content to its customers in less time. This scenario illustrates the use of XML for content management.
- A company analyzes and stores questionnaire response information in XML format to help them respond to customer needs better. This scenario can be classified as the use of XML for content management.

How SQL Server 2008 Implements XML



Key Points

Microsoft SQL Server 2008 supports several techniques for querying and analyzing data stored in XML format.

- Relational data can be transformed to XML by using the For XML clause in T-SQL.
- Data that already exists in XML format can be parsed and transformed to relational data by using the OpenXML function.
- XQuery is based on the existing XPath query language, with support added for better iteration, better sorting results, and the ability to construct the necessary XML. XQuery operates on the XQuery Data Model.
- Before you can create typed xml variables, parameters, or columns, you must first register the XML schema collection by using CREATE XML SCHEMA COLLECTION (Transact-SQL). You can then associate the XML schema collection with variables, parameters, or columns of the xml data type.

The XML Data Type

The **XML** data type is a built-in data type in SQL Server® for storing and interacting with XML data

The XML data type supports five methods for interacting with the data that it contains

Query	Accepts XQuery statements to retrieve elements of the XML data
Value	Retrieves a value of SQL type from an XML instance
Exists	Determines if an XQuery statement returns results
Modify	Is used to specify XML data modification update statements
Nodes	Splits XML into multiple rows

Key Points

The XML data type is a built-in data type for storing and interacting with XML data. It has five methods that provide the ability to query, modify, and alter the data that it contains.

- The Query, Value, Exists, and Nodes methods accept XQuery statements.
- The Modify method accepts XML Data Modification Language statements.

Lesson 3

Querying XML Data

- Using For XML to Generate XML
- Querying XML by Using OpenXML
- Querying XML Using XQuery
- Generating XML-Based Reports

In this lesson, you will learn the different techniques that can be used to generate XML format from SQL Server 2008 data as well as how to query data that already exists in XML format.

Using FOR XML to Generate XML

```
SELECT Cust.CustomerID,
       OrderHeader.CustomerID,
       OrderHeader.SalesOrderID,
       OrderHeader.Status,
       Cust.CustomerType
  FROM Sales.Customer Cust, Sales.SalesOrderHeader OrderHeader
 WHERE Cust.CustomerID = OrderHeader.CustomerID
 ORDER BY Cust.CustomerID
FOR XML AUTO {RAW('ElementName') | EXPLICIT}
```

```
<Cust CustomerID="1" CustomerType="S">
  <OrderHeader CustomerID="1" SalesOrderID="43860" Status="5" />
  <OrderHeader CustomerID="1" SalesOrderID="44501" Status="5" />
  <OrderHeader CustomerID="1" SalesOrderID="45283" Status="5" />
  <OrderHeader CustomerID="1" SalesOrderID="46042" Status="5" />
</Cust>
```

Key Points

The FOR XML clause in T-SQL can be used to generate an XML representation of any rowset.

- For XML supports extensions to provide control over the resulting XML structure.
- Three modes – Raw, Auto, and Explicit – control the format of the resulting XML.
- The Root extension can be used to provide a root node path for the resulting XML.
- The Path mode can be used to define the XML path syntax.

Querying XML by Using OpenXML

```
DECLARE @xml_text VARCHAR(4000), @i INT  
  
SELECT @xml_text =  
'<root><person LastName="White" FirstName="Johnson"/>  
<person LastName="Green" FirstName="Marjorie"/>  
<person LastName="Carson" FirstName="Cheryl"/></root>'  
  
EXEC sp_xml_preparedocument @i OUTPUT, @xml_text  
  
SELECT * FROM  
OPENXML(@i, '/root/person') WITH (LastName nvarchar(50),  
FirstName nvarchar(50))  
  
EXEC sp_xml_removedocument @i
```

Last Name	First Name
White	Johnson
Green	Marjorie
Carson	Cheryl

Key Points

OPENXML can be used in SELECT and SELECT INTO statements wherever rowset providers, a view, or OPENROWSET can appear as the source.

- OPENXML, a Transact-SQL keyword, provides a rowset for in-memory XML documents that is similar to a table or a view.
- The system stored procedure sp_xml_preparedocument must be called to prepare the XML data for reading prior to using OpenXML.
- OpenXML allows the row and column XPath patterns to be parameterized as variables.

Querying XML Using XQuery

XQuery defines the **FLWOR** iteration syntax. FLWOR is the acronym for for, let, where, order by, and return.

```
SELECT Instructions.query('*'
    declare namespace AWMI="http://schemas.microsoft.com
    /sqlserver/2004/07/adventureworks/ProductModelManuInstructions";
    for $T in //AWMI:tool
        let $L := //AWMI:Location[./AWMI:tool[.=data($T)]]
        return
            <tool desc="{data($T)}" Locations="{data($L/@LocationID)}"/>
') as Result
FROM Production.ProductModel
where ProductModelID=7
```

Result

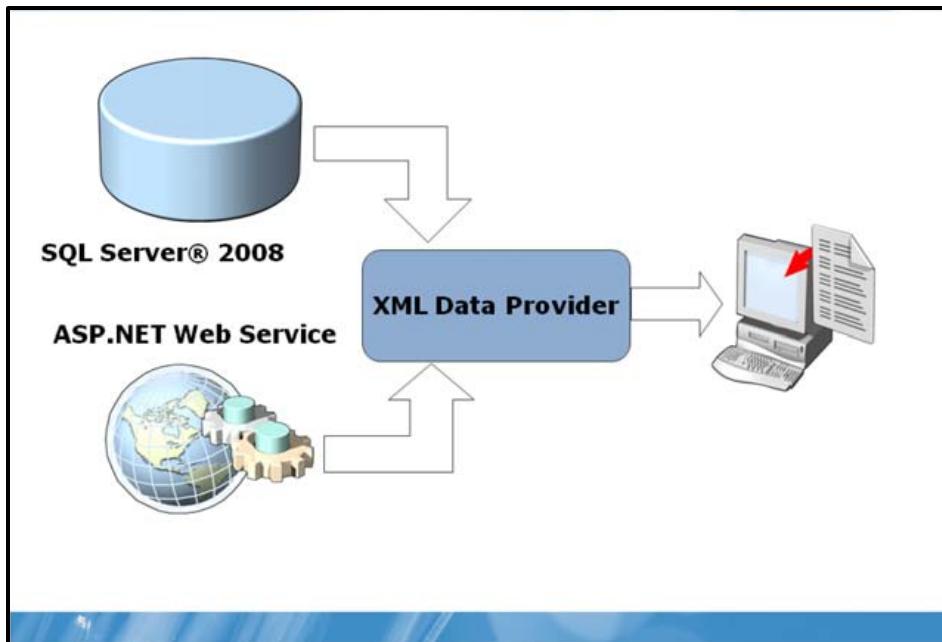
```
<tool desc="hammer" Locations="30"/>
```

Key Points

XQuery is a query language designed and developed for querying data stored in XML.

- The query method of the XML SQL Server data type accepts XQuery statements.
- The results of the XQuery statements from the query method are XML data type fragments.

Generating XML-Based Reports



Key Points

The XML data provider is a Microsoft® .NET data provider that is provided with Reporting Services.

- The XML content can be embedded directly within the query and the developer can build queries and data dynamically within the report.
- XML content can also be accessed directly from a URL.
- An XML data provider can query Web services directly by parsing the XML structure of the SOAP response directly.

Demonstration: Using XML

- Using For XML to Generate XML Results
- Using OpenXML to Query XML
- Using XQuery to Query XML

Question: Why would XQuery be used in a query?

Question: Why would OpenXML be used?

Lesson 4

Overview of Full-Text Indexes

- What Are Full-Text Indexes?
- How Full-Text Indexes Are Populated
- Full-Text Indexing and Querying Process
- How to Implement Full-Text Indexes in SQL Server® 2008

In this lesson, you will learn about full-text indexes including what full-text indexes are and how they are set up. The session also explores how SQL Server 2008 implements and supports full-text indexes.

What Are Full-Text Indexes?

Definition: A full-text index is a special type of token-based functional index that provides extended searching capabilities for text data.

Use Cases

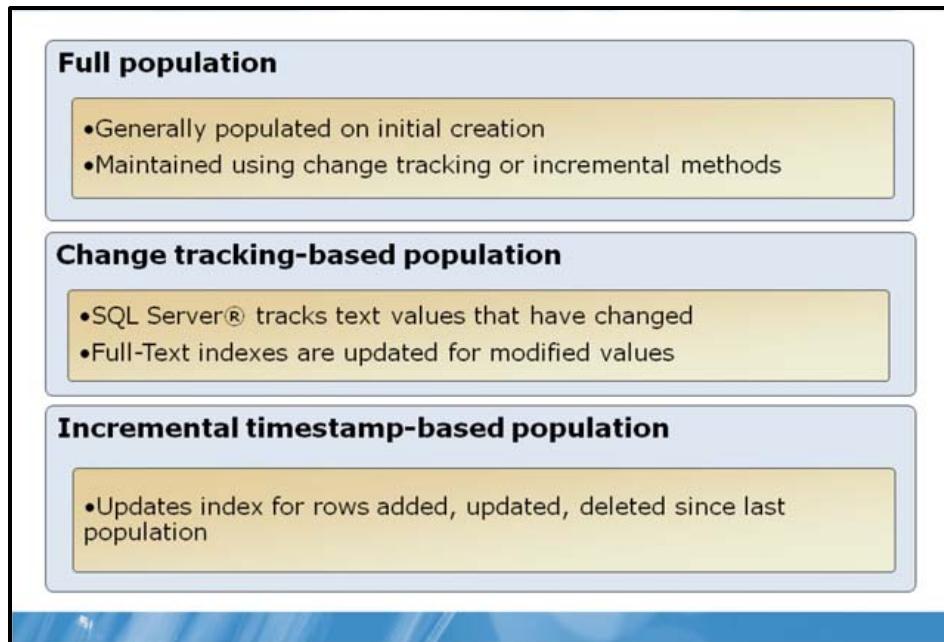
- Exposing data for advanced web site searches
- Allowing fuzzy searches of product descriptions
- Allowing wildcard searches of customer addresses

Key Points

A full-text index is a special type of token-based functional index that is built and maintained by the Microsoft Full-Text Engine for SQL Server (MSFTESQL) service.

- From web site searches to fuzzy searches of customer data, there are many cases for the use of full-text indexes.

How Full-Text Indexes Are Populated

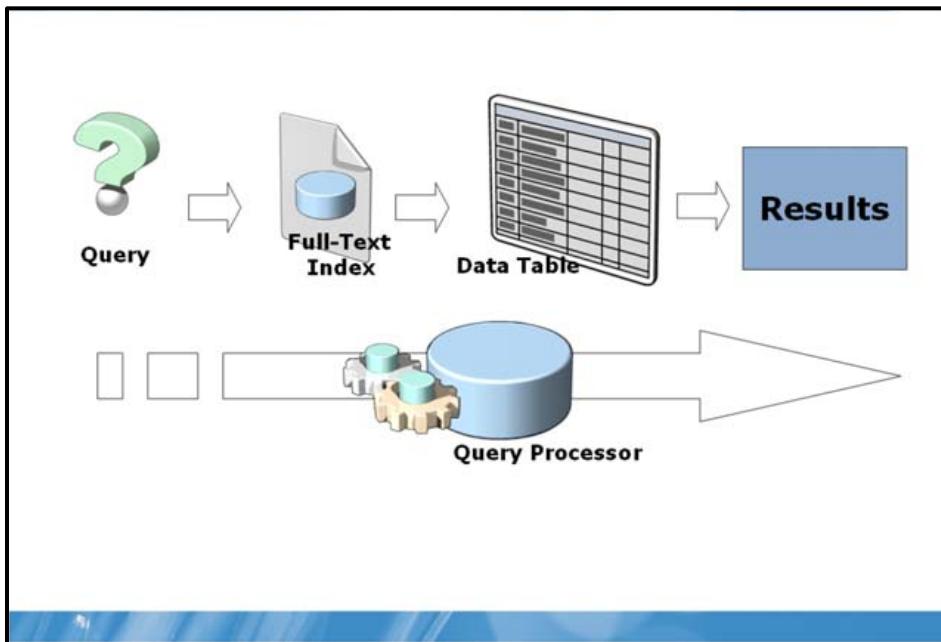


Key Points

The process of creating and maintaining a full-text index is called index population.

- Full population is a complete creation of the index and is generally only used when the index is initially created.
- Change tracking-based population involves SQL Server tracking what values have changed.
- Incremental timestamp-based population requires a timestamp column in the indexed table that is used to determine rows that have been added, updated, or deleted.
- Change tracking and Incremental timestamp based populations can be set to update the index manually or automatically on a schedule.

Full-Text Indexing and Querying Process



Key Points

Querying a table using a full-text index is a process that involves several steps.

- A full-text query sent from a client goes to the SQL Server Query Processor (QP) in the SQL Server process.
- The QP passes this on to the full-text query component, which creates an OLE DB command tree and sends it on to the Microsoft Full-Text Engine for the SQL Server (MSFTESQL) service.
- In the MSFTESQL process, the Full-Text Engine Query Processor processes the query using thesaurus, noise word files, word breakers and stemmers.
- After processing this query, the MSFTESQL service returns a result set to the SQL Server process.
- A result set can be used for further processing or returned to the client.

How to Implement Full-Text Indexes in SQL Server 2008

- Determine the tables and columns that require full-text indexing according to business requirements
- Enable full-text indexing in the database
 - Use sp_fulltext_database 'enable'
- Create the full-text index for the required table(s)
 - Use sp_fulltext_table
- Add required column(s) to the index
 - Use sp_fulltext_column
- Activate and set population options for index
 - Use sp_fulltext_table
- Design and build queries using full-text query functions and predicates according to business requirements

Key Points

SQL Server 2008 provides system stored procedures that can be used to enable, set up, configure, modify, activate, disable, and destroy full-text indexes. Full-text indexes can also be set up and administered via features built in to SQL Server® Management Studio.

- The first step is to clearly define the tables and columns that need full-text indexes based on business requirements.
- The full-text indexes needed can then be created, set up, and enabled using either system stored procedures or SQL Server Management Studio.
- Finally, the queries can be created to use the full-text indexes in order to satisfy business requirements.

Lesson 5

Querying Full-Text Indexes

- Overview of Full-Text Search
- The CONTAINS Predicate
- The FREETEXT Predicate
- Full-Text Functions
- Combining Full-Text Search and T-SQL Predicates

In this lesson, you will learn how full-text indexes can be used within a query. First, the T-SQL predicates CONTAINS and FREETEXT is explored. Next, the use of full-text functions is discussed.

Overview of Full-Text Search

Exact Search: WHERE sentence LIKE '%run %'

Results:

- We had to **run** in gym class
- The politician decided to **run** for office

Full-Text Search

Results:

- We had to **run** in gym class
- The politician decided to **run** for office
- My car is not **running**
- He had to rest after he **ran** a mile
- Doctors say **running** is a healthy hobby



Key Points

Full-text queries can include words and phrases, or multiple forms of a word or phrase and allows fast and flexible indexing for keyword-based query of text data stored in a SQL Server database.

- In SQL Server 2008, full-text search delivers enterprise-level search functionality.
- Full-text search in SQL Server 2008 can scale from small mobile or personal deployments with relatively few and simple queries, up to complex mission-critical applications with high query volume over large quantities of textual data.

The CONTAINS Predicate

```
SELECT Name  
FROM Production.Product  
WHERE CONTAINS(Name, ' "Chain*"');
```

Sample CONTAINS conditions

- CONTAINS(Name, ' "Mountain" OR "Road" ')
- CONTAINS(Description, 'bike NEAR performance')
- CONTAINS(Description, ' FORMSOF (INFLECTIONAL, ride) ')
- CONTAINS(Description, 'ISABOUT (performance weight (.8),
comfortable weight (.4), smooth weight (.2))')

Key Points

CONTAINS offers a complex syntax for using a full-text indexed column.

- CONTAINS accepts several keywords that provides very flexible searching:
 - FORMSOF – search for different forms of the root word.
 - NEAR – search for one word that is near another word.
 - ISABOUT ... weight – search for multiple words and provide a weighting for each word's importance.
 - OR – search for multiple words that only require one match.
 - * - use an asterisk for a wildcard match.

The FREETEXT Predicate

```
SELECT Title  
FROM Production.Document  
WHERE FREETEXT (Document, 'vital safety components' );
```

Steps that FREETEXT takes:

- Word-breaking: "vital", "safety", "components"
- Stemming: "vital", "safe", "safety", "components"
- List of expansions: "vital", "important", "safe", "safety", "components", "parts"

Key Points

The FREETEXT predicate is used to search columns containing character-based data types for values that match the meaning and not the exact wording of the words in the search condition.

- The full-text query engine internally performs the following actions on the freetext_string, assigns each term a weight, and then finds the matches.
 - Separates the string into individual words based on word boundaries (word-breaking).
 - Generates inflectional forms of the words (stemming).
 - Identifies a list of expansions or replacements for the terms based on matches in the thesaurus in queries.

Full-Text Functions

```
SELECT Product.ProdID, Product.Name, Prod.Desc, Keys.Rank  
FROM dbo.Product  
INNER JOIN FREETEXTTABLE(dbo.Product, Desc,  
    'safety harness',LANGUAGE 'English',2) AS Keys  
ON Product.ProdID = Keys.[KEY];
```

Key Table

Key	Rank

Product Table

ProdID	Name	Desc

Key Points

Full-Text functions – CONTAINSTABLE and FREETEXTTABLE – provide the same searching capabilities as their Full-Text predicate counterparts, CONTAINS and FREETEXT.

- The functions return matches in table form.
- The tables that are returned from the functions contain two columns – Key and Rank.
- The Key column contains the primary key value from the matching rows and the Rank column contains a value indicating the quality of the match for each row.
- The functions may return zero, one, or multiple matching rows.

Combining Full-Text Search and T-SQL Predicates

```
SELECT Product.ProductDescriptionID, Product.Description, Keys.Rank
FROM Production.ProductDescription Product
INNER JOIN FREETEXTTABLE (Production.ProductDescription,
[Description],
'safety',LANGUAGE 'English',2) AS Keys
ON Product.ProductDescriptionID = Keys.[KEY]
WHERE Product.QuantityAvailable > 0;
```

ProductDescriptionID	Description	Rank
513	All-occasion value bike with our basic comfort and safety features. Offers wider, more stable tires for a ride around town or weekend trip.	134
594	Travel in style and comfort. Designed for maximum comfort and safety. Wide gear range takes on all hills. High-tech aluminum alloy construction provides durability without added weight.	67

Key Points

In most cases, just a single Full-Text search will not yield the exact result set that you are looking for in the search.

- Most queries will also involve other T-SQL conditions.

Demonstration: Full-Text Index

- Creating a Full Text Index
- Querying a Full Text Index Using Predicates
- Querying a Full Text Index Using Functions

Question: Why would you choose to use a full text search function rather than a predicate?

Question: Why would you choose to use a full text search predicate rather than a function?

Lab: Querying Metadata, XML, and Full-Text Indexes

- Exercise 1: Querying Metadata
- Exercise 2: Querying XML data
- Exercise 3: Creating and Querying Full Text Indexes

Logon information

Virtual machine	NY-SQL-01
User name	Administrator
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Exercise 1: Querying Metadata

Scenario

You are the database administrator at Adventure Works. The company requires that you perform an annual audit of the database systems and their efficiency. As part of the audit, you need to verify the metadata elements of the AdventureWorks database.

The main tasks for this exercise are as follows:

1. Launch **SQL Server Management Studio**.
2. Query metadata using system catalog views to get a list of column names for the Person table.
3. Query metadata using the information schema to get column information for the Address table.
4. Query metadata using dynamic management views to get information about recently executed commands.
5. Query metadata using system stored procedures to get column information for the Employee table.

► **Task 1: Launch SQL Server Management Studio**

- Start the **2778A-NY-SQL-01** virtual machine, and log on as **Administrator** with the password of **Pa\$\$w0rd**.
- Open **SQL Server Management Studio**.

► **Task 2: Query metadata using System Catalog Views**

- Create a query that will return the column name from sys.columns.
- Use a Join to join the sys.columns view with the sys.tables view on object_id. Use a WHERE clause to restrict the result set to the Person table.
- Execute the query and review the results.

► **Task 3: Query metadata using the Information Schema**

- Create a query that will return all values from the INFORMATION_SCHEMA.Columns view for the Address table.
- Execute the query and review the results.

► **Task 4: Query metadata using Dynamic Management Views**

- Create a query that will return all values from the sys.dm_exec_requests view.
- Refine the query to group the results by the command name, count the number of occurrences of each, and return the average total elapsed time of each command.
- Execute the query and review the results.

► **Task 5: Query metadata using System Stored Procedures**

- Create a query that executes the sp_columns stored procedure and send 'Employee' to the @table_name parameter and 'HumanResources' to the @table_owner parameter.
- Execute the query and review the results.

Results: After this exercise, you should have launched SQL Server Management Studio and queried metadata using system catalog views, the information schema, dynamic management views, and system stored procedures.

Exercise 2: Querying XML Data

Scenario

One important task includes the retrieving of the orders data in an XML format from an external system. To do this, you need to query the required orders data and retrieve it in an XML format.

The main tasks for this exercise are as follows:

1. Create XML output using For XML.
2. Query XML data using OpenXML.

► Task 1: Create XML output using For XML

- Create a query that will return the CustomerID, SalesOrderID, and Status columns from the `Sales.SalesOrderHeader` table.
- Refine query by ordering by CustomerID and specify the output to be in XML by using For XML Auto.
- Execute the query and review the results.

► Task 2: Query XML data using OpenXML

- Open the predefined query at `E:\Mod07\Labfiles\Exercise2Task2`.
- Add a call to `sp_xml_preparedocument` to prepare the XML for the query.
- Execute the query and review the results.

Results: After this exercise, you should have created a query of relational data and produced the output in XML format. You should have also created a query that converts data in XML format to a relational format.

Exercise 3: Creating and Querying Full-Text Indexes

Scenario

The company requires your help to create search functionality for products. The web site development team is developing an enhanced search feature for the online catalog. To support that functionality, you need to implement a full-text catalog for the product descriptions.

The main tasks for this exercise are as follows:

1. Create a full-text index for the product description column.
2. Query the full-text index using FREETEXT.
3. Query the full-text index using CONTAINS.

► Task 1: Create a full-text index

- Use the New Full Text Catalog dialog in SQL Server Management Studio to create a new full-text index named ProductDescriptionCatalog.
- Edit the properties of ProductDescriptionCatalog to include the Production.ProductDescription.Description column.
- Set a schedule for the catalog to be populated every hour.
- Rebuild the catalog.

► Task 2: Query the full-text index using FREETEXT

- Create a query that lists all products from Production.ProductDescription where the Description column contains a form of the word "lightest".
- Execute the query and review the results.

► Task 3: Query the full-text index using CONTAINS

- Create a query that lists all products from Production.ProductDescription where the Description column contains a form of the word "lightest" which is near the word "best".
- Execute the query and review the results.

Results: After this exercise, you should have created a new full-text catalog and queried that catalog using both the FREETEXT and CONTAINS predicates.

Lab Shutdown

After you complete the lab, you must shut down the **2778A-NY-SQL-01** virtual machine and discard any changes.

Module Review and Takeaways

- **Review Questions**

Review Questions

1. What type of functionality in an application requires querying of metadata?
2. What method would be used to convert data in XML format to a relational format?
3. What is the difference between a full-text query predicate and function?

Module 8

Using Programming Objects for Data Retrieval

Contents:

Lesson 1: Overview of Views	8-3
Lesson 2: Overview of User-Defined Functions	8-13
Lesson 3: Overview of Stored Procedures	8-21
Lesson 4: Overview of Triggers	8-27
Lesson 5: Writing Distributed Queries	8-33
Lab: Using Programming Objects for Data Retrieval	8-38

Module Overview

- Overview of Views
- Overview of User-Defined Functions
- Overview of Stored Procedures
- Overview of Triggers
- Writing Distributed Queries

Developing applications using Microsoft® SQL Server® 2008 involves the use of programming objects such as views, user-defined functions, and stored procedures. These objects provide logical structure for queries and data modification methods, are capable of processing business logic, and enforcing business rules. This module provides an overview of these objects, demonstrate their capabilities, and provide best practices regarding their use.

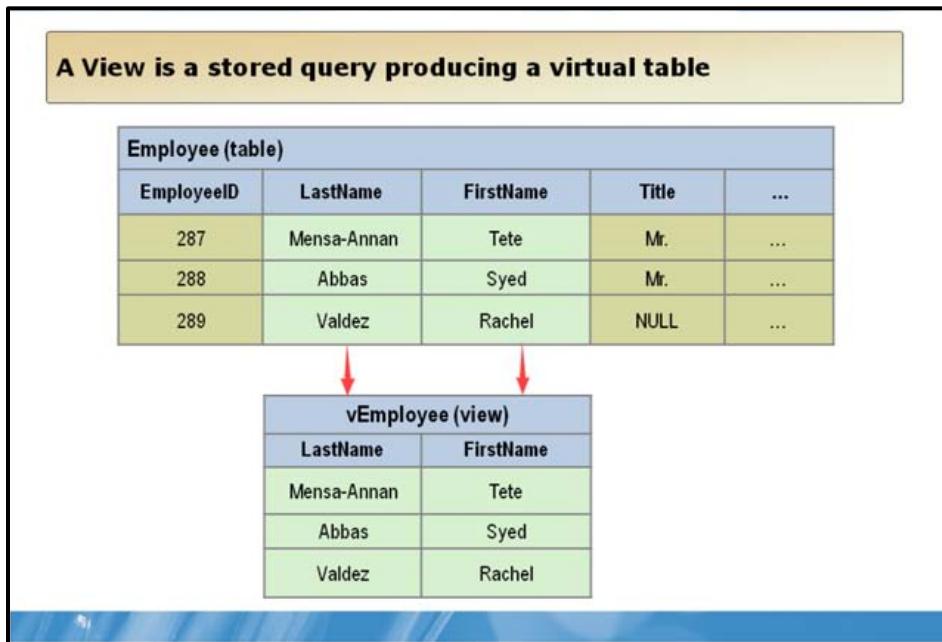
Lesson 1

Overview of Views

- What Are Views?
- Creating and Managing a View
- Considerations When Creating Views
- Restrictions for Modifying Data by Using Views
- Indexed Views
- Indexed View Example
- Partitioned Views
- Partitioned View Example

In this lesson, you will learn what views are and how they can be used to encapsulate queries.

What Are Views?



Key Points

A view is used to do any or all of these functions:

- Restrict a user to specific rows in a table. For example, allow an employee to see only the rows recording his or her work in a labor-tracking table.
- Restrict a user to specific columns. For example, allow employees who do not work in payroll to see the name, office, work phone, and department columns in an employee table, but do not allow them to see any columns with salary information or personal information.
- Join columns from multiple tables so that they look like a single table.
- Aggregate information instead of supplying details. For example, present the sum of a column, or the maximum or minimum value from a column.

Creating and Managing a View

Creating or Modifying a view

```
[CREATE|ALTER] VIEW HumanResources.vEmployee  
AS  
BEGIN  
  
SELECT EmployeeID, FirstName, LastName,  
EmailAddress  
FROM HumanResources.Employee  
  
END
```

Deleting a view

```
DROP VIEW HumanResources.vEmployee
```

Key Points

T-SQL Statements can be used to create, modify, or delete a view:

- You can create a view by using the CREATE VIEW statement. A view can be created only in the current database and it can have a maximum of 1024 columns.
- You can modify the definition of an existing view by using the ALTER VIEW statement. You can alter views without affecting dependent stored procedures or triggers and without changing permissions. You can also alter indexed views.
- You can remove one or more views from the current database by using the DROP VIEW statement.
- After a view has been created, it can be queried just like a normal table. When you execute a query on a view, the Database Engine checks that all the database objects referenced in the statement exist and that they are valid in the context of the statement.

Considerations When Creating Views

Restriction	Description
Column Limit	<ul style="list-style-type: none"> Total number of columns referenced in the view cannot exceed 1024
COMPUTE	<ul style="list-style-type: none"> Cannot be used in a CREATE VIEW definition
ORDER BY	<ul style="list-style-type: none"> Cannot be used in views, inline functions, derived tables, and subqueries
INTO	<ul style="list-style-type: none"> Cannot be used with the SELECT statement in a view definition
Temporary table	<ul style="list-style-type: none"> Cannot be referenced in a view
GO	<ul style="list-style-type: none"> CREATE VIEW must be alone in a single batch
SELECT *	<ul style="list-style-type: none"> Can be used in a view definition if the SCHEMABINDING clause is not specified

Key Points

The following restrictions must be considered when creating views:

- The total number of columns referenced in the view cannot exceed 1024 due to the maximum capacity specification of SQL Server.
- You cannot use the COMPUTE clause in a CREATE VIEW definition.
- You cannot use the ORDER BY clause in views.
- You cannot use the INTO clause with the SELECT statement in a view definition.
- You cannot define a view that references a temporary table or table variable.
- You cannot combine the CREATE VIEW statement with any other statement in a single batch. Therefore, you must specify a GO statement before and after any CREATE PROCEDURE, CREATE VIEW, or CREATE FUNCTION commands to isolate them from the rest of the instructions in a batch.
- You can use SELECT * in a view definition as long as the SCHEMABINDING clause is not specified in the view.

Restrictions for Modifying Data by Using Views

Restrictions to writing statements that modify data:

- Statements must modify columns from only one base table
- Follow criteria when WITH CHECK OPTION is used
- INSERT must specify values for all non-nullable columns

Restrictions to columns when modifying data:

- Table columns must be referenced directly
- Use an INSTEAD OF trigger
- Cannot be affected by GROUP BY, HAVING, or DISTINCT clauses.
- Data must follow the restrictions on the modified columns

Key Points

The following restrictions must be considered when modifying data using views:

- Any modifications, such as UPDATE, INSERT, and DELETE statements, when applied to a view, must modify columns from only one base table.
- If WITH CHECK OPTION is used in the view definition, all data modification statements executed against the view must adhere to the criteria set within the SELECT statement defining the view.
- INSERT statements must specify values for any columns in the underlying table that do not allow null values and have no DEFAULT definitions.
- Columns being modified in a view must reference the underlying data in the table columns directly.
- You cannot modify columns computed or derived by using aggregate functions such as AVG, COUNT, SUM, MIN, MAX, GROUPING, STDEV, STDEVP, VAR, and VARP.
- Columns computed from an expression or set operators such as UNION, UNION ALL, CROSSJOIN, EXCEPT, and INTERSECT cannot be modified unless you specify an INSTEAD OF trigger.

Indexed Views

An Indexed View is a view for which a unique clustered index has been created.

Indexed View Details

- Views can be indexed using CREATE INDEX
- Should not be used for views whose underlying data is updated frequently
- Columns must be listed explicitly
- Views must be created with the SCHEMABINDING option

Key Points

Unique clustered indexes can be created for views to improve performance.

- If views are frequently referenced in queries, you can improve performance by creating a unique clustered index on the view.
- When a unique clustered index is created on a view, the result set is stored in the database just like a table with a clustered index is stored.
- The index can improve query performance of the view.

Indexed View Example

Creating an Indexed View

```
CREATE VIEW vwDiscount WITH SCHEMABINDING AS  
  
SELECT SUM(UnitPrice*OrderQty)AS SumPrice,  
SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount))AS  
SumDiscountPrice,  
SUM(UnitPrice*OrderQty*UnitPriceDiscount)AS  
SumDiscountPrice2,  
COUNT_BIG(*) AS Count, ProductID  
FROM Sales.SalesOrderDetail  
GROUP BY ProductID  
  
GO  
  
CREATE UNIQUE CLUSTERED INDEX vwDiscountInd ON vwDiscount  
(ProductID)
```

Key Points

- To be able to index a view, the view must be defined using the WITH SCHEMABINDING option.
- The view must be written to have a unique identifier for the index to be created on.
- The index is created using the CREATE UNIQUE CLUSTERED INDEX statement.

Partitioned Views

A partitioned view joins horizontally partitioned data from a set of member tables across one or more servers, making the data appear as if from one table.

Partitioned View Details

- Allows the data in a large table to be split into smaller member tables.
- Data can be partitioned between the member tables based on ranges of data values.
- Partitioned Views make it easier to maintain the member tables independently.

Key Points

Partitioning data can help distribute the load of large tables and improve the query performance of that data.

- A partitioned view joins horizontally partitioned data from a set of member tables across one or more servers, making the data appear as if from one table.
- Partitioned Views allow the data in a large table to be split into smaller member tables.
- Data can be partitioned between the member tables based on ranges of data values.
- Partitioned Views make it easier to maintain the member tables independently.
- Partitioned Views use the UNION ALL statement to combine the rows from the member tables.

Partitioned View Example

Creating a Partitioned View

```
CREATE TABLE May1998Sales
(OrderID INT,
 CustomerID INT NOT NULL,
 OrderDate DATETIME NULL
CHECK (DATEPART(yy,
OrderDate) = 1998),
 OrderMonth INT
CHECK (OrderMonth = 5),
 DeliveryDate DATETIME NULL
CONSTRAINT OrderIDMonth
PRIMARY KEY(OrderID,
OrderMonth)
)
```

```
CREATE VIEW Year1998Sales
AS
SELECT * FROM Jan1998Sales
UNION ALL
SELECT * FROM Feb1998Sales
UNION ALL
SELECT * FROM Mar1998Sales
UNION ALL
SELECT * FROM Apr1998Sales
UNION ALL
SELECT * FROM May1998Sales
UNION ALL
SELECT * FROM Jun1998Sales
UNION ALL
SELECT * FROM Jul1998Sales
UNION ALL
...

```

Key Points

- Partitioned tables are useful for dividing the load of large tables into a series of smaller tables.
- Partitioned tables use check constraints to define the range of data that should exist in each table.
- The partitioned view uses the UNION ALL operator to combine the partitioned tables into a single result set.

Demonstration: Building a View

In this demonstration, you will see how to:

- Create a view
- Query a view
- Generate a script for a view

Question: Why would an indexed view be used?

Question: Why would a partitioned view be used?

Lesson 2

Overview of User-Defined Functions

- What Are User-Defined Functions?
- Creating and Managing User-Defined Functions
- Creating a Table Valued User-Defined Function
- Restrictions When Creating User-Defined Functions
- How to Implement Different Types of User-Defined Functions
- Performance Considerations for Using User-Defined Functions

In this lesson, you will learn about User-Defined Functions. The topics will include how they are created, what different types are possible, restrictions of user-defined functions, and how they can affect query performance.

What Are User-Defined Functions?

A User-Defined Function is a routine that accepts parameters, performs an action, and returns the result of that action as a value.

Benefits of using User-Defined Functions

- Modular programming for reusable logic.
- Complex operations can be optimized for faster execution.
- Logic performed in database reduces network traffic

Key Points

User-Defined Functions are structures that can contain business logic that is applied to data stored in the database:

- User-defined functions can be used to encapsulate common logic that needs to be accessible from multiple areas.
- User-defined functions in Microsoft SQL Server 2008 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.
- All user-defined functions contain a two-part structure of a header and a body.
- SQL Server 2008 provides three types of user-defined functions, namely, scalar functions, table-valued functions, and built-in functions.

Creating and Managing User-Defined Functions

Creating or Modifying a User-Defined Function

```
[CREATE|ALTER] FUNCTION fEmployeeEmail(@ID int)
RETURNS varchar(50)
AS
BEGIN
DECLARE @email varchar(50)

SELECT @email = EmailAddress
FROM HumanResources.Employee
WHERE EmployeeID = @ID

RETURN @email
END
```

Deleting a view

```
DROP FUNCTION fEmployeeEmail
```

Key Points

T-SQL statements can be used to create, modify, or delete user-defined functions:

- In SQL Server 2008, user-defined functions can be created by using the CREATE FUNCTION statement.
- Existing user-defined functions can be modified by using the ALTER FUNCTION statement.
- User-defined functions can be removed by using the DROP FUNCTION statement.
- Each fully qualified user-defined function name (schema_name.function_name) must be unique.
- User-defined functions can be invoked in queries, statements, or expressions.

Creating a Table Valued User-Defined Function

Creating a Table Valued User-Defined Function

```
CREATE FUNCTION fEmployeeByGender(@Gender nchar(1))
RETURNS table
AS
BEGIN

RETURN (SELECT *
       FROM HumanResources.Employee
      WHERE Gender = @Gender)

END
```

Querying a Table Valued User-Defined Function

```
SELECT * FROM fEmployeeByGender('F')
```

Key Points

- A table valued user-defined function is specified by using RETURNS table in the function definition.
- The RETURN statement is used to return the rows from the function.
- The table valued user-defined function can be queried using a SELECT... FROM statement just like a table.

Restrictions When Creating User-Defined Functions

You cannot use user-defined functions to:

- Update data
 - Use stored procedures instead
- Define or create new objects in the database
 - Objects referred to by the function have to be previously declared and created
- Perform transactions

Key Points

Some restrictions exist regarding the contents of a user-defined function:

- User-defined functions cannot be used to update data. You can use stored procedures instead.
- User-defined functions cannot define or create new objects in the database.
- All objects referred to by the function, with the exception of scalar types, have to be previously declared and created.
- Transactions cannot be performed inside a user-defined function.

How to Implement Different Types of User-Defined Functions

Types	Usage
Scalar-valued	<ul style="list-style-type: none">Scalar is specified in the RETURNS clauseCan be defined with multiple T-SQL statements
Inline table-valued	<ul style="list-style-type: none">TABLE is specified in the RETURNS clauseDoes not have associated return variables<i>select_stmt</i> is the single SELECT statement that defines the return value
Multi-statement table-valued	<ul style="list-style-type: none">TABLE is specified in the RETURNS clause<i>function_body</i> is used as a series of T-SQL statements that populate a TABLE return variable<i>@return_variable</i> is used to store and accumulate rows that are returned as the value

Key Points

User-defined functions are either scalar-valued or table-valued.

- User-defined functions are scalar-valued if SCALAR is specified in the RETURNS clause.
- Scalar-valued functions can be defined by using multiple T-SQL statements.
- Functions are table-valued if TABLE is specified in the RETURNS clause.
- Depending on how the body of the function is defined, table-valued functions can be classified as inline table-valued functions or multi-statement table-valued functions.

Performance Considerations for Using User-Defined Functions

```
SELECT MyCalculation(column_name)
FROM table_name
WHERE MyCondition(column_name)
```

- Both functions will be called once for each row in the table.
- If both functions require 0.1 second for each execution, the query will require:
 - 1 second for 5 rows
 - 10 seconds for 50 rows
 - 1 hour for 18,000 rows

Key Points

When evaluating user-defined functions, it is vital to consider the balance between performance and maintainability.

- If a user-defined function is used in a SELECT or a WHERE clause it will be executed for every row.
- Function performance should be considered along with the number of rows passed through it.

Demonstration: Building a User-Defined Function

In this demonstration, you will see how to:

- Create a user-defined function
- Call a user-defined function

Question: Why would a scalar type user-defined function be used?

Question: Why would a table type user-defined function be used?

Lesson 3

Overview of Stored Procedures

- What Are Stored Procedures?
- How Are Stored Procedures Created?
- Stored Procedure Initial Execution
- Stored Procedure Practices

In this lesson, stored procedures and triggers will be defined. You will learn how they are created, how they work, and what purposes they serve.

What Are Stored Procedures?

A collection of T-SQL statements stored on the server

Stored Procedures Can:

- Accept input parameters
- Return output parameters or rowset
- Return a status value to indicate success or failure

Benefits of using Stored Procedures:

- Promotes modular programming
- Provides security attributes and permission chaining
- Allows delayed binding and code reuse
- Reduces network traffic

Key Points

A stored procedure is a named collection of T-SQL statements that is stored on the server.

- You can encapsulate repetitive tasks that execute efficiently by using a stored procedure.
- Stored procedures accept input parameters and return output parameters to the calling procedure.
- Stored procedures contain programming statements which perform operations on a database.
- Stored procedures also return a status value to a calling procedure to indicate success or failure.

How Are Stored Procedures Created?

Creating a Stored Procedure

```
CREATE PROCEDURE HumanResources.usp_GetEmployeesName
@NamePrefix char(1)
AS
BEGIN
SELECT BusinessEntityID, FirstName, LastName,
EmailAddress
FROM HumanResources.vEmployee
WHERE FirstName LIKE @NamePrefix + '%'
ORDER BY FirstName
END
```

Calling a Stored Procedure

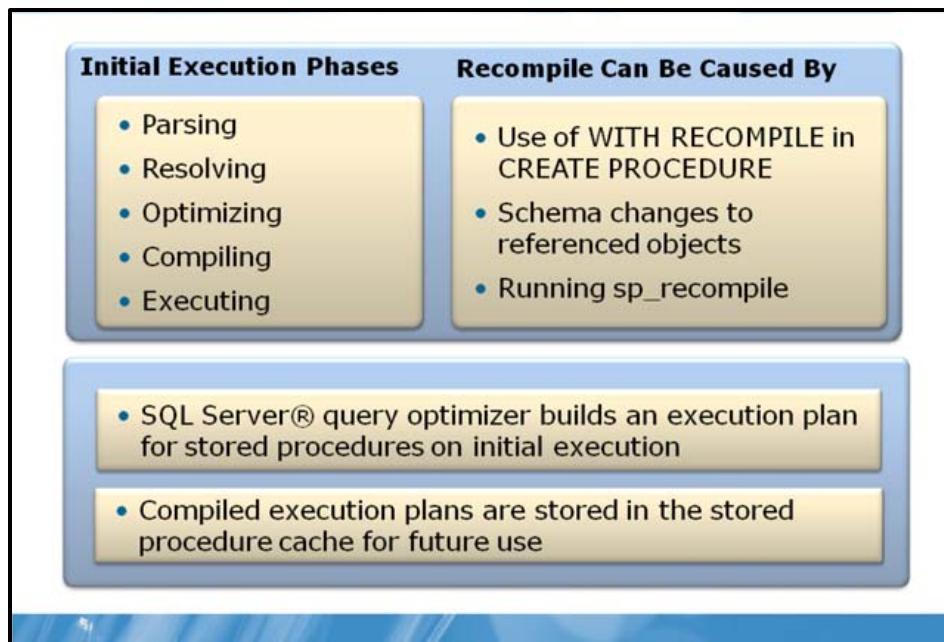
```
EXECUTE HumanResources.usp_GetEmployeesName 'A'
```

Key Points

Stored procedures are created using the CREATE PROCEDURE statement.

- Stored procedures can contain input and output parameters.
- The body of a stored procedure can contain T-SQL statements, logical statements, and transactions.
- Stored procedures are invoked using the EXECUTE statement.

Stored Procedure Initial Execution



Key Points

There are a number of steps performed when a Stored Procedure is executed for the first time:

- When you execute a stored procedure for the first time, the SQL Server query optimizer builds an execution plan for the stored procedure.
- The compiled execution plan is stored in the stored procedure cache, in memory, on the SQL Server.
- SQL Server tries to reuse this plan for subsequent executions of the stored procedure.

Stored Procedure Tips and Best Practices

Stored Procedure Tips

- Use WITH ENCRYPTION to hide procedure source
- Use WITH RECOMPILE to force recompilation on each execution

Stored Procedure Best Practices

- Validate all input parameters
- Avoid building string based SQL within procedure to reduce the risk of SQL injection
- Use cursors sparingly

Key Points

The following practices should be considered when designing and writing stored procedures:

- The WITH ENCRYPTION clause of the CREATE PROCEDURE statement will cause the procedure source to be obfuscated so that it cannot be retrieved in its original form directly from metadata.
- When input parameters are accepted, they should always be validated before use to maintain security.
- Cursors are very useful tools but are resource intensive.

Demonstration: Creating Stored Procedures

In this demonstration, you will see how to:

- Create a stored procedure
- Call a stored procedure

Question: How do Stored Procedures promote modular programming?

Lesson 4

Overview of Triggers

- What Are Triggers?
- How Are Triggers Created?
- How Triggers Work
- Trigger Types and Limitations

In this lesson, triggers will be defined. You will learn how they are created, how they work, and what purposes they serve.

What Are Triggers?

A collection of T-SQL statements stored on the server that Evaluates data before or after it is inserted, modified, or Deleted.

- A special type of stored procedure that executes when an attempt is made to modify data in a table
- Typically used to maintain low-level data integrity and not used to return query results
- Two categories of DML triggers: INSTEAD OF and AFTER. INSTEAD OF trigger is also known as the BEFORE trigger

Key Points

A trigger is a special type of stored procedure that executes whenever an attempt is made to modify data in a table that the trigger protects.

- Triggers are typically used to maintain low-level data integrity and not used to return query results.
- There are two categories of DML triggers: INSTEAD OF and AFTER. The INSTEAD OF trigger is also known as the BEFORE trigger.
- INSTEAD OF triggers are executed in place of the triggering action.
- AFTER triggers are executed after the action of the INSERT, UPDATE, or DELETE statement is performed.

How Are Triggers Created?

Creating a Trigger

```
CREATE TRIGGER Sales.trigCurrency  
  ON Sales.Currency  
  AFTER INSERT  
  AS  
  BEGIN  
    DECLARE @name nvarchar(50)  
    SELECT @name = Name  
    FROM inserted  
    IF len(@name) < 5  
    BEGIN  
      ROLLBACK TRANSACTION  
    END  
END
```

Key Points

- Triggers are created using the CREATE TRIGGER statement.
- The CREATE TRIGGER statement specifies the name of the trigger, the table that the trigger should reference, and the type of the trigger.
- Triggers can be defined as "AFTER" meaning that it executes after the specified action or "INSTEAD OF" which executes before the specified action.
- Triggers can be created for the INSERT, UPDATE, or DELETE actions.

How Triggers Work

- Triggers can roll back transactions if a specific business rule is not satisfied
- When a trigger that contains a rollback statement is executed from an SQL batch, the entire batch is canceled
- Any statement following the ROLLBACK TRANSACTION statement will still be executed
- Any modifications that happen after the rollback are not rolled back

Key Points

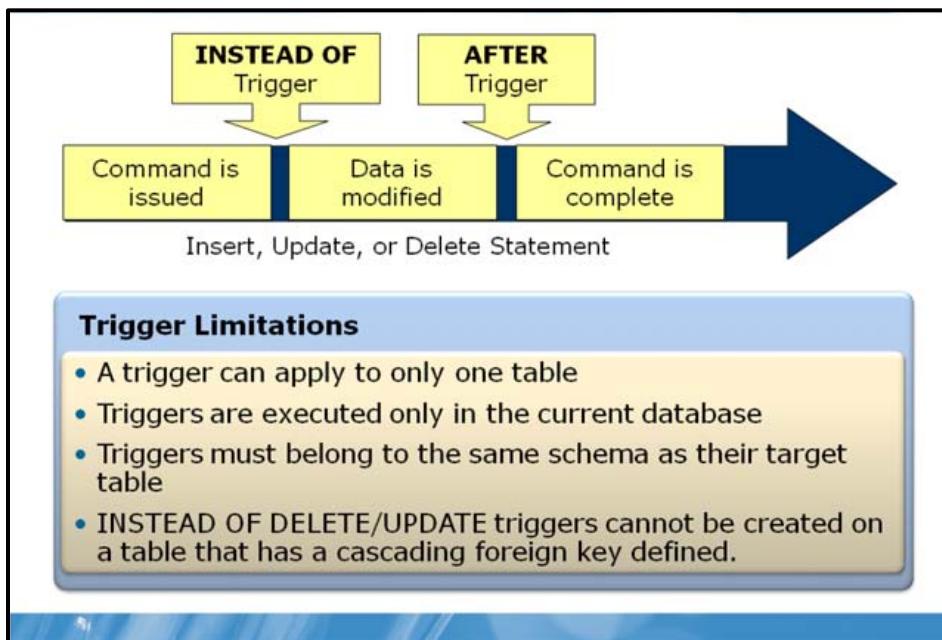
Triggers can roll back transactions if a specific business rule is not satisfied.

- When a trigger that contains a rollback statement is executed from an SQL batch, the entire batch is canceled.
- All the data that was modified by the triggering action is rolled back by the ROLLBACK TRANSACTION statement of the trigger.
- Any statement following the ROLLBACK TRANSACTION statement will still be executed.



Note: It is generally recommended to avoid giving any statements after a ROLLBACK TRANSACTION statement, unless they are informational.

Trigger Types and Limitations



Key Points

- INSTEAD OF triggers execute after the specified command is issued but before the data in the table is modified.
- AFTER triggers execute after the data has been modified.
- Each trigger can apply to only one trigger.
- Triggers must belong to the same schema as their target table.

Demonstration: Creating Triggers

In this demonstration, you will see how to:

- Create a trigger
- See the results of a trigger

Question: Why would you use an INSTEAD OF trigger rather than an AFTER trigger?

Question: Why would you use an AFTER trigger rather than an INSTEAD OF trigger?

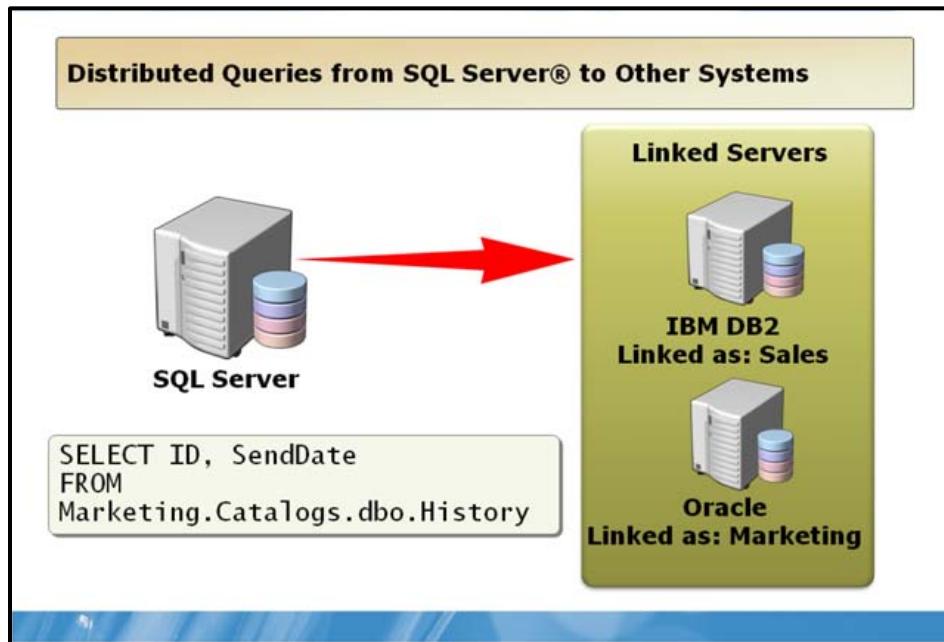
Lesson 5

Writing Distributed Queries

- How SQL Server Works with Heterogeneous Data
- Adding Linked Servers
- Using Ad Hoc Distributed Queries
- How to Write Linked Server-Based Distributed Queries

In today's business system architecture, data is stored in various locations in various formats. Using SQL Server 2008, you can access heterogeneous data from a wide variety of systems and in a wide variety of formats. This lesson will focus on how heterogeneous data can be accessed in SQL Server.

How SQL Server Works With Heterogeneous Data



Key Points

- Distributed queries can allow users to access another data source (for example, files, non-relational data sources such as Active Directory, and so on) using the security context of the Microsoft Windows® account under which the SQL Server service is running. SQL Server 2008 impersonates the login appropriately for Windows logins.
- Use the sp_addlinkedserver stored procedure to define the specific logins that are authorized to access the corresponding linked server. This control is not available for ad hoc names, so use caution in enabling an OLE DB provider for ad hoc access.

Adding Linked Servers

The diagram illustrates the process of adding linked servers in SQL Server. It features two main sections of T-SQL code on the left and two associated steps on the right.

T-SQL Code:

```
Use AdventureWorks ;  
GO  
EXEC sp_addlinkedserver  
    @server = 'Marketing',  
    @srvproduct = 'Oracle',  
    @provider = 'MSDAORA',  
    @datasrc = 'OraServer'
```



```
Use AdventureWorks ;  
GO  
EXEC sp_addlinkedsrvlogin  
    @rmtsrvname = 'Marketing',  
    @useself = 'false',  
    @locallogin = 'Mary',  
    @rmtuser = 'MaryP',  
    @rmtpassword = 'd89q3w4u'
```

Associated Steps:

- Add a Linked Server**
 - Other Databases
 - Files
 - Non-Relational Data Sources
- Map Credentials**
 - Associate with linked server
 - Specify local login
 - Map to remote login

Key Points

- sp_addlinkedserver creates a linked server. A linked server allows for access to distributed, heterogeneous queries against OLE DB data sources. After a linked server is created by using sp_addlinkedserver, distributed queries can be run against this server. If the linked server is defined as an instance of SQL Server, remote stored procedures can be executed.
- When a user logs on to the local server and executes a distributed query that accesses a table on the linked server, the local server must log on to the linked server on behalf of the user to access that table. Use sp_addlinkedsrvlogin to specify the login credentials that the local server uses to log on to the linked server.

Using Ad Hoc Distributed Queries

- Accesses remote heterogeneous data
- Best if the remote data does not need to be accessed often
- A permanent link is not required for better security

```
SELECT ID, SendDate  
FROM OPENDATASOURCE('MSDAORA',  
'Data Source=OraServer; User=MaryP;  
Password=d89q3w4u').Catalog.dbo.History'
```

```
SELECT ID, SendDate  
FROM OPENROWSET('MSDAORA', 'OraServer', 'MaryP',  
'd89q3w4u', 'SELECT * FROM Catalog.dbo.History')
```

Key Points

You can access remote heterogeneous data by writing ad hoc distributed queries.

- This technique is best used if the remote data does not need to be accessed frequently.
- By using ad hoc distributed queries, you can avoid setting up a permanent link to the external data source and thus maintain better security.
- SQL Server provides two rowset functions, such as OPENROWSET and OPENDATASOURCE, which can be used for writing ad hoc distributed queries.

How to Write Linked Server-Based Distributed Queries

- Use a fully qualified four-part name and the OPENQUERY function
- The OPENQUERY function executes the query on the specified linked server

```
SELECT *
FROM OPENQUERY(OracleSvr, 'SELECT name, id FROM
HumanResources.Titles')
```

Key Points

Linked servers provide consistent connections to remote data sources.

- Queries can be written against data in linked servers by using either a fully qualified four-part name or by using the OPENQUERY function.
- The four-part name consists of linked server name, the catalog, the schema name, and finally the name of the object to be queried.
- The OPENQUERY function accepts the name of the linked server to be queried as well as the SQL to be submitted to that linked server.

Lab: Using Programming Objects for Data Retrieval

- Exercise 1: Creating Views
- Exercise 2: Creating User-Defined Functions
- Exercise 3: Creating Stored Procedures
- Exercise 4: Writing Distributed Queries

Logon information

Virtual machine	NY-SQL-01
User name	Administrator
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Exercise 1: Creating Views

Scenario

As the database administrator at Adventure Works, you need to create views which present a simplified version of tables with only the relevant columns from the AdventureWorks database. The Person table contains a lot of information but in most cases only the name is required. In this exercise, you will create a view that will return only the name and ID elements from that table.

The main task for this exercise are as follows:

- Create a view to select only the name and ID columns from the Person table.
- **Task 1: Create a view to select only the name and ID columns from the Person table**
- In **SQL Server Management Studio**, create a new view.
 - In the view, select BusinessEntityID, FirstName, MiddleName, and LastName from the **Person.Person** table.
 - Create the view and query it.

Results: After this exercise, you should have created a view and queried it using a T-SQL query.

Exercise 2: Creating User-Defined Functions

Scenario

You are the database administrator at Adventure Works. You need to create several user-defined functions to meet the custom requirements of your organization. You will create a scalar user-defined function that removes the time part from the datetime() object and present only the date related information. Yet another user-defined function retrieves details about purchase orders of a particular vendor when supplied with a VendorID.

The main tasks for this exercise are as follows:

1. Launch **SQL Server Management Studio**.
2. Create a scalar valued user-defined function that removes the time part from a datetime value.
3. Create a table valued user-defined function that retrieves purchase orders by VendorID.

► Task 1: Launch SQL Server Management Studio

- Start the **2778A-NY-SQL-01** virtual machine, and log on as **Administrator** with the password of **Pa\$\$w0rd**.
- Open **SQL Server Management Studio**.

► Task 2: Create a scalar valued user-defined function

- In the AdventureWorks2008 database, create a new scalar valued user-defined function that returns varchar(10) and accepts a datetime input parameter.
- In the body of the function, perform the logic to remove the time portion from the datetime parameter value. For information on datetime functions, refer to SQL Server® Books Online and search for DATEPART.
- Create the function and test it by passing the current date to it in a T-SQL query. The current datetime value can be obtained by using the GETDATE() function.

► Task 3: Create a table valued user-defined function

- In the AdventureWorks2008 database, create a new table valued user-defined function that accepts an int input parameter as the VendorID.
- In the body of the function, query the Purchasing.PurchaseOrderHeader table using the VendorID in the parameter.
- Create the function and test it by passing the VendorID 1624 to it in a T-SQL query.

Results: After this exercise, you should have launched SQL Server Management Studio and created and tested both a scalar valued and table valued user-defined function.

Exercise 3: Creating Stored Procedures

Scenario

As the database administrator at Adventure Works, you need to create stored procedures which perform common tasks. The Person table is queried often to retrieve a person's name using the ID column. In this exercise, you will create a stored procedure that will retrieve a person's name using the ID.

The main tasks for this exercise are as follows:

1. Create a stored procedure to select the name columns from the Person table for a given ID.

► **Task 1: Create a stored procedure to select the name columns from the Person table for a given ID**

- In **SQL Server Management Studio**, create a new stored procedure.
- Accept an integer PersonID value as an input parameter.
- In the stored procedure, select FirstName, MiddleName, and LastName from the view created in Exercise 2.
- Create the stored procedure.

Results: After this exercise, you should have created a stored procedure that queries a view.

Exercise 4: Writing Distributed Queries

Scenario

The Adventure Works sales system produces a monthly sales summary report in Microsoft® Office Excel® format. This Excel file is placed on your database server every time it is created. You will execute an ad hoc distributed query against the data in that spreadsheet. Finally, you will create a linked server to that spreadsheet and query the linked server.

The main tasks for this exercise are as follows:

1. Enable distributed queries on the server.
2. Create and execute an ad hoc distributed query against the spreadsheet.
3. Create a linked server to the spreadsheet.
4. Query the linked spreadsheet.

► Task 1: Enabling distributed queries

- Execute the following statements:

```
sp_configure 'show advanced options', 1
reconfigure

sp_configure 'Ad Hoc Distributed Queries', 1
reconfigure.
```

► Task 2: Executing an ad hoc distributed query against a Microsoft Office Excel spreadsheet

- Create a query that uses the OPENROWSET function with the following parameters:

```
'Microsoft.Jet.OLEDB.4.0',
'Excel 8.0;Database=E:\Mod08\Labfiles\SalesSummary2008.xls',
'SELECT Country, TotalSales FROM [Sheet1$]'
```

► Task 3: Create a linked server to a Microsoft Office Excel spreadsheet

- Execute the sp_addlinkedserver system stored procedure with the following parameters:

```
'SalesSummary',
'Jet 4.0',
'Microsoft.Jet.OLEDB.4.0',
'E:\Mod08\Labfiles\SalesSummary2008.xls',
NULL,
'Excel 8.0'
```

► Task 4: Query the linked Microsoft Office Excel spreadsheet

- Create a query for the 'Sheet1\$' table of the 'SalesSummary' linked server using a four part name.
- Execute the query and review the results.

Results: After this exercise, you should have executed an ad hoc distributed query and created and queried a linked server.

Lab Shutdown

After you complete the lab, you must shut down the **2778A-NY-SQL-01** virtual machine and discard any changes.

Module Review and Takeaways

• Review Questions

Review Questions

1. What option must be specified in a CREATE VIEW in order for the view to be indexable?
2. How can the source of a stored procedure be hidden?
3. What special kind of stored procedure can be used to roll back an attempted data modification in a SQL Server table?
4. What types of data sources can be linked to SQL Server 2008 for distributed queries?

Module 9

Using Advanced Techniques

Contents:

Lesson 1: Considerations for Querying Data	9-3
Lesson 2: Working with Data Types	9-11
Lesson 3: Cursors and Set-Based Queries	9-19
Lesson 4: Dynamic SQL	9-26
Lesson 5: Maintaining Query Files	9-31
Lab: Using Advanced Techniques	9-34

Module Overview

- Considerations for Querying Data
- Working with Data Types
- Cursors and Set-Based Queries
- Dynamic SQL
- Maintaining Query Files

This module provides you with the skills and knowledge about advanced querying techniques in Microsoft® SQL Server® 2008. This course provides information about the best practices you should follow when querying complex data that involves date/time data, and how to work with the hierarchyid data type. You will learn how to use cursors and set-based queries, as well as dynamic SQL. In addition, you will examine how to use Microsoft® Team Foundation Server for managing query source files.

Lesson 1

Considerations for Querying Data

- Execution Plans
- Data Type Conversions
- Implicit Conversions
- Explicit Conversions with CAST and CONVERT
- Data Type Precedence

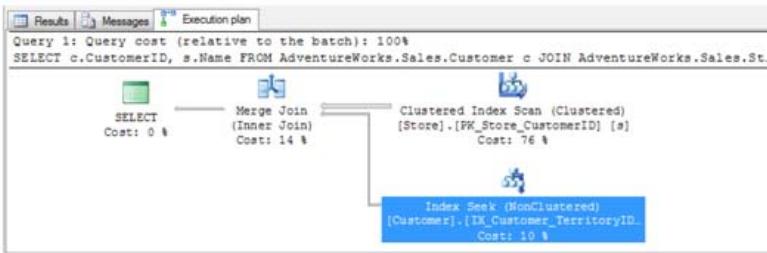
Because there are often many ways to query for the same data, you need to understand how SQL Server processes queries using execution plans to help identify the most efficient query methods for a given situation. Also, when querying data, you need to understand how data type conversion works, as well as the order of precedence of data types.

Execution Plans

- Shows how the Database Engine navigates tables and indexes
- View Estimated Execution Plan before query execution
- View Actual Execution Plan after execution

Generate Execution Plans with:

- SQL Server Management Studio
- T-SQL SET options
- SQL Server Profiler



Key Points

Execution plans graphically display the data retrieval methods chosen by the SQL Server query optimizer.

- Execution plans represent the execution cost of specific statements and queries in SQL Server using icons.
- To use the graphical execution plan feature in Management Studio, and to use the Showplan Transact-SQL SET statement options, users must have sufficient permissions to execute the Transact-SQL statements and queries. Users must also be granted the SHOWPLAN permission for all databases containing referenced objects.
- The graphical execution plan output in SQL Server Management Studio is read from right to left and from top to bottom. Each query in the batch that is analyzed is displayed, including the cost of each query as a percentage of the total cost of the batch.



Note: For more information about the icons used to display execution plans in Management Studio, see the topic Graphical Execution Plan Icons (SQL Server Management Studio) in SQL Server Books Online.

Question: Why would you want to view the Estimated Execution Plan?

Demonstration: Using Execution Plans

In this demonstration, you will learn how to:

- View estimated and actual execution plans

Question: What do you need to do in order to be able to view the Actual Execution Plan?

Data Type Conversions

Data Type Conversion scenarios

- Data is moved to, compared, or combined with other data
- Data is moved from a result column, return code, or output parameter into a program variable

Implicit Conversion

- Transparent to the user

Explicit Conversion

- Uses CAST or CONVERT

Key Points

Data types can be converted in the following scenarios:

- When data from one object is moved to, compared with, or combined with data from another object, the data may have to be converted from the data type of one object to the data type of the other.
- When data from a Transact-SQL result column, return code, or output parameter is moved into a program variable, the data must be converted from the SQL Server system data type to the data type of the variable.

Implicit conversions are not visible to the user. SQL Server automatically converts the data from one data type to another. For example, when a **smallint** is compared to an **int**, the **smallint** is implicitly converted to **int** before the comparison proceeds.

Explicit conversions use the CAST or CONVERT functions. The CAST and CONVERT functions convert a value (a local variable, a column, or another expression) from one data type to another. For example, the following CAST function converts the numeric value of \$157.27 into a character string of '157.27':

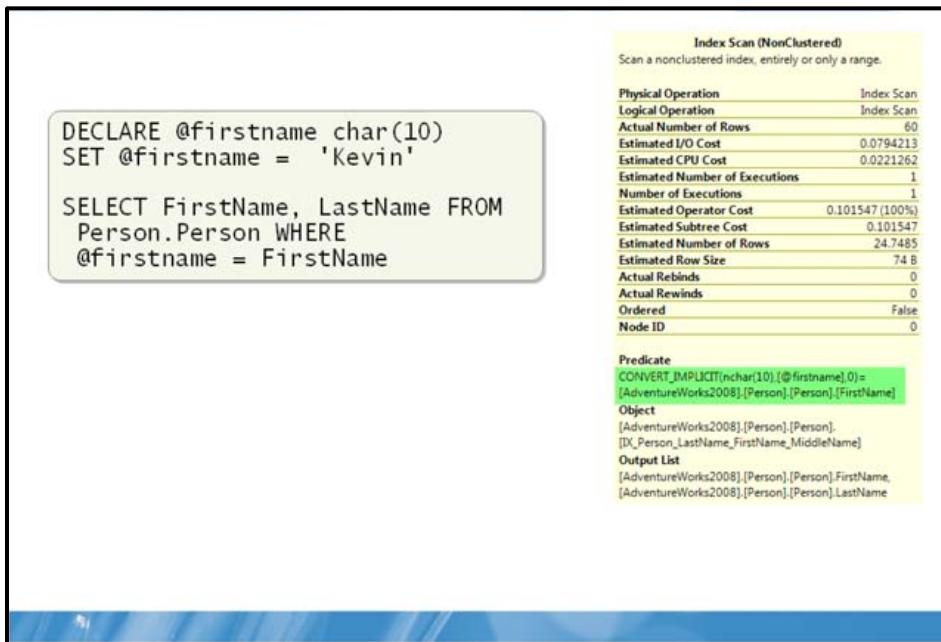
```
CAST ( $157.27 AS VARCHAR(10) )
```



Note: Not all data type conversions are allowed. For a chart of allowed implicit and explicit conversions, see the topic CAST and CONVERT (Transact-SQL) in SQL Server Books Online.

Question: What are some examples of operations that can result in data conversion?

Implicit Conversions



Key Points

Implicit conversions occur whenever two different data types are operated upon together without the use of CAST or CONVERT. In the code sample above, the variable **@firstname** is a **char** data type. The **FirstName** field in the **Person.Person** table is an **nvarchar** data type.

In this case, **char** has a lower precedence than **nvarchar**, so the **@firstname** variable is converted to **nvarchar** for comparison, as the execution plan shows. Data type precedence is covered shortly.

Question: When would you want to avoid implicit conversions?

Explicit Conversions with CAST and CONVERT

Using CAST

```
USE AdventureWorks2008;
GO
SELECT SUBSTRING(Name, 1, 30) AS ProductName, ListPrice
FROM Production.Product
WHERE CAST(ListPrice AS int) LIKE '3%';
GO
```

Using CONVERT

```
USE AdventureWorks2008;
GO
SELECT SUBSTRING(Name, 1, 30) AS ProductName, ListPrice
FROM Production.Product
WHERE CONVERT(int, ListPrice) LIKE '3%';
GO
```

Key Points

- CAST and CONVERT explicitly convert one data type to another, with slightly different syntaxes.
- CONVERT is SQL Server specific and includes additional styles for converting date/time data.
- Use CAST instead of CONVERT if you want Transact-SQL program code to comply with ISO. Use CONVERT instead of CAST to take advantage of the style functionality in CONVERT.

Data Type Precedence

Without explicit conversion, this statement fails

```
DECLARE @label varchar(12),
        @pageno int
SET @label='Page Number '
SET @pageno = 1
Print @label + @pageno
```

With explicit conversion, this statement succeeds

```
DECLARE @label varchar(12),
        @pageno int
SET @label='Page Number '
SET @pageno = 1
Print @label + CONVERT(varchar, @pageno)
```

Key Points

When an operator combines two expressions of different data types, the rules for data type precedence specify that the data type with the lower precedence is converted to the data type with the higher precedence. If the conversion is not a supported implicit conversion, an error is returned. When both operand expressions have the same data type, the result of the operation has that data type.

In the code sample above, the first statement fails because **int** has higher precedence than **varchar**. The string '**Page Number**' cannot be converted to **int**, so an explicit conversion of the **@pageno** variable to **varchar** is required for the statement to succeed and produce the correct output.



Note: For a complete list of data type precedence, see the topic Data Type Precedence (Transact-SQL) in SQL Server Books Online.

Question: Can you think of other scenarios where data type precedence might cause problems for implicit conversions?

Demonstration: Understanding Data Type Conversion

In this demonstration, you will learn how to:

- Understand explicit and implicit data type conversions
- Use CAST and CONVERT to explicitly convert data types

Question: How can implicit data type conversions cause unexpected results on queries that run successfully?

Lesson 2

Working with Data Types

- Recommendations for Querying Date/Time Data
- Recommendations for Inserting Date/Time Data
- Implementing the hierarchyid Data Type
- Working with Hierarchies

With the introduction of new date/time data types in SQL Server 2008, you need to know how to work with these data types in order to make sure your time/date data is correct in the database and in reports. Another new data type, hierarchyid, can help organize hierarchical data in your databases.

Recommendations for Querying Date/Time Data

- Date/Time values can be queried using numeric operators such as =, >, and < as well as date/time functions.
- When querying date/time data, care must be taken in understanding the data type.

DATETIME, DATETIME2, & DATETIMEOFFSET data types

- **Query conditions must include both date and time portions.**

DATE data type

- **Query conditions must include just the date portion.**

TIME data type

- **Query conditions must include just the time portion.**

Key Points

In SQL Server, there are six built-in data types to store date and time data: **time**, **date**, **smalldatetime**, **datetime**, **datetime2**, and **datetimeoffset**.

The **smalldatetime**, **datetime** and **datetime2** data types store both date and time data. The **datetimeoffset** data type also stores both date and time data, but is time-zone aware for specifying the offset from a time or datetime value. The **time** and **date** data types store only time and only date data, respectively.

If you specify only the date part when inserting data to **smalldatetime**, **datetime**, or **datetime2** column, then SQL Server stores a zero value in the time part. If you specify only the time part, then SQL Server stores 1900-01-01 in the date part. Because the date and time values are stored together in those data types, some unexpected problems might occur.

The following table summarizes the key differences between the different date and time data types:

Data type	Format	Range	Accuracy	Storage size (bytes)	User-defined fractional second precision	Time zone offset
time	hh:mm:ss[.nnnnnnn]	00:00:00.0000000 through 23:59:59.9999999	100 nanoseconds	3 to 5	Yes	No
date	YYYY-MM-DD	0001-01-01 through 9999-12-31	1 day	3	No	No
smalldatetime	YYYY-MM-DD	1900-01-01	1 minute	4	No	No

Data type	Format	Range	Accuracy	Storage size (bytes)	User-defined fractional second precision	Time zone offset
	hh:mm:ss	through 2079-06-06				
datetime	YYYY-MM-DD hh:mm:ss[.nnn]	1753-01-01 through 9999-12-31	0.00333 second	8	No	No
datetime2	YYYY-MM-DD hh:mm:ss[.nnnnnnn]	0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999	100 nanoseconds	6 to 8	Yes	No
datetimeoffset	YYYY-MM-DD hh:mm:ss[.nnnnnnn] [+ -]hh:mm	0001-01-01 00:00:00.0000000 through 9999-12-31 23:59:59.9999999 (in UTC)	100 nanoseconds	8 to 10	Yes	Yes

Recommendations for Inserting Date/Time Data

- Use correct format and language settings
- Use language independent formats for portability

```
SET DATEFORMAT mdy
GO
DECLARE @datevar datetime
SET @datevar = '12/31/2008'
SELECT @datevar
```

```
SET DATEFORMAT ydm
GO
DECLARE @datevar datetime
SET @datevar = '2008/31/12'
SELECT @datevar
```

```
SET DATEFORMAT ymd
GO
DECLARE @datevar datetime
SET @datevar = '2008/12/31'
SELECT @datevar
```

Key Points

When inserting date/time values into tables or views, you need to ensure that the format and the language settings of the date/time data types are correct. It is recommended that you use language-independent formats, rather than language-dependent formats because they are portable across languages. It is not recommended that you use language-dependent formats even with SET statements.

You can also set SQL Server to use the ISOdatetime format. In the ISO format, the datetime values are represented as yyyyymmdd [hh:mm:ss] or yyyy-mm-ddThh:mm:ss. The ISO dateformat is always guaranteed to work irrespective of the LANGUAGE or DATEFORMAT setting.

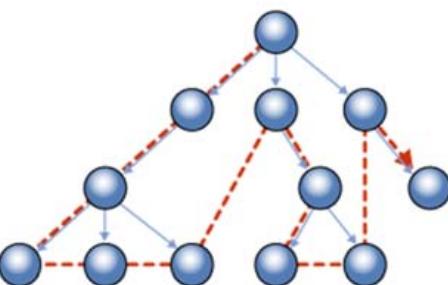
Demonstration: Working with Date/Time Data

In this demonstration, you will learn how to:

- Use the correct methods for working with various date/time data types

Question: What happens when you insert data containing only a date or only a time into a column that holds date and time data?

Implementing the hierarchyid Data Type



```
CREATE TABLE Organization
(
    EmployeeID hierarchyid,
    OrgLevel as EmployeeID.GetLevel(),
    EmployeeName nvarchar(50) NOT NULL
)
GO
```

Key Points

Hierarchical data is defined as a set of data items that are related to each other by hierarchical relationships. Hierarchical relationships are where one item of data is the parent of another item. Hierarchical data is common in databases. Examples include the following:

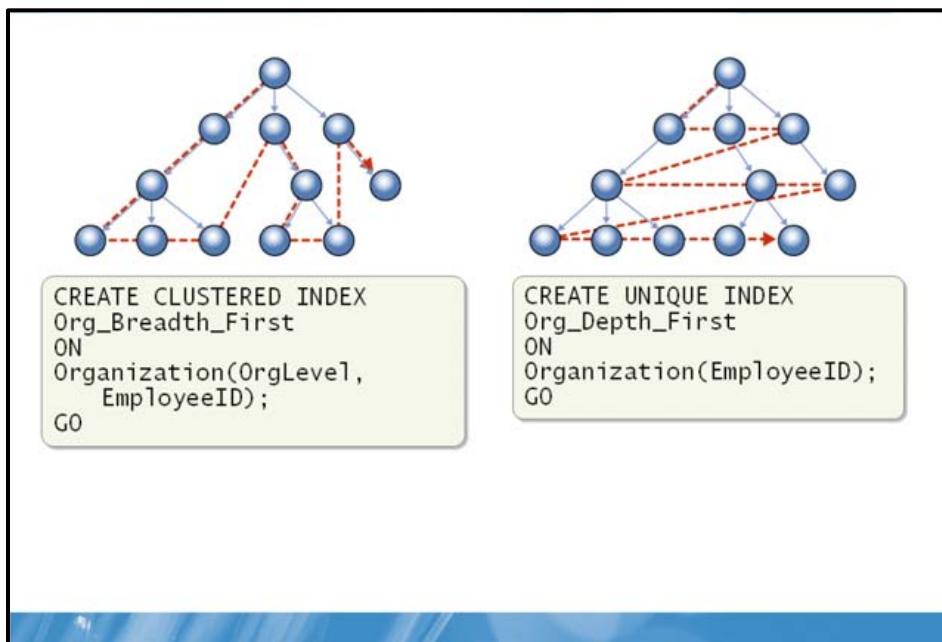
- An organizational structure
- A file system
- A set of tasks in a project
- A taxonomy of language terms
- A graph of links between Web pages

The **hierarchyid** data type makes it easier to store and query hierarchical data. **hierarchyid** is optimized for representing trees, which are the most common type of hierarchical data. The **hierarchyid** data type can be used to replace recursive common table expressions (CTEs) which are frequently used to generate hierarchical representations of database data.

The code sample shows how to create a table that implements the **hierarchyid** data type on the **EmployeeID** column for organizing employees in the database in a hierarchy.

Question: Can you think of a scenario where using hierarchical data might be useful?

Working with Hierarchies



Key Points

A hierarchy needs an indexing strategy. There are two indexing strategies supported:

- Depth-first: A depth-first index, rows in a sub-tree are stored near each other. For example, all employees that report through a manager are stored near their managers' record. This is represented in the code sample on the left.
- Breadth-first: A breadth-first stores the rows each level of the hierarchy together. For example, the records of employees who directly report to the same manager are stored near each other. This is represented by the code sample on the right.

You can perform a number of operations on nodes within the hierarchy. You can query to get nodes above or below a point on the tree, move nodes from one parent to another, and add and remove nodes using the methods supplied with the hierarchyid data type.



Note: For a complete list and description of methods available for working with hierarchies, see the topic hierarchyid Data Type Method Reference in SQL Server Books Online.

Demonstration: Using the hierarchyid Data Type

In this demonstration you will learn how to:

- Work with the hierarchyid data type

Question: What is the difference between the **GetDescendant** and **GetAncestor** methods?

Lesson 3

Cursors and Set-Based Queries

- Understanding Cursors
- Cursor Implementations
- Using Cursors
- Understanding Set-Based Logic

When querying data, the set of rows returned from a query is called the result set. Cursors allow you to perform operations on the result set on a row-by-row basis. Set-based logic, or a set-based approach, deals with the result set as a whole or parts instead of individual rows of data.

Understanding Cursors

Cursors extend processing of result sets

- Allow positioning at specific rows
- Retrieve one or more rows from the current position
- Support data modification
- Support different levels of visibility
- Provide T-SQL statements access to data

Key Points

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.

Question: Can you think of a scenario where a cursor might prove useful?

Cursor Implementations

Implementation	Features
Transact-SQL	<ul style="list-style-type: none">• Based on DECLARE CURSOR• Implemented on server
API Server	<ul style="list-style-type: none">• Based on API functions in OLE DB and ODBC• Implemented on server
Client	<ul style="list-style-type: none">• Implemented by Native Client ODBC and ADO• Caches all result set rows on the client

Key Points

SQL Server supports three cursor implementations.

- **Transact-SQL cursors**

Are based on the DECLARE CURSOR syntax and are used mainly in Transact-SQL scripts, stored procedures, and triggers. Transact-SQL cursors are implemented on the server and are managed by Transact-SQL statements sent from the client to the server. They may also be contained in batches, stored procedures, or triggers.

- **Application programming interface (API) server cursors**

Support the API cursor functions in OLE DB and ODBC. API server cursors are implemented on the server. Each time a client application calls an API cursor function, the SQL Server Native Client OLE DB provider or ODBC driver transmits the request to the server for action against the API server cursor.

- **Client cursors**

Are implemented internally by the SQL Server Native Client ODBC driver and by the DLL that implements the ADO API. Client cursors are implemented by caching all the result set rows on the client. Each time a client application calls an API cursor function, the SQL Server Native Client ODBC driver or the ADO DLL performs the cursor operation on the result set rows cached on the client.



Note: Because Transact-SQL cursors and API server cursors are implemented on the server, they are referred to collectively as server cursors.

Using Cursors

Process of using a cursor

- Associate and define characteristics
- Populate the cursor
- Retrieve rows in the cursor
- Modify data if needed
- Close and deallocate the cursor

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

Key Points

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

- 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.
- Execute the Transact-SQL statement to populate the cursor.
- 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.
- Optionally, perform modification operations (update or delete) on the row at the current position in the cursor.
- Close the cursor.

When you are completely finished with a cursor, you must also deallocate the resources using the DEALLOCATE command. Closing a cursor releases locks, but SQL Server resources will be held until a cursor is deallocated.

Demonstration: Working with Cursors

In this demonstration, you will learn how to:

- Use cursors to work with result sets

Question: How do you specify that a cursor should be read only?

Understanding Set-Based Logic

Set-based logic

- **SQL Server iterates through data**
- **Deals with results as a set instead of row-by-row**

```
SELECT ProductID,
Purchasing.Vendor.VendorID, Name
FROM Purchasing.ProductVendor JOIN
Purchasing.Vendor
ON (Purchasing.ProductVendor.VendorID
= Purchasing.Vendor.VendorID)
WHERE StandardPrice > $10
AND Name LIKE N'F%'
GO
```

Key Points

While cursors allow a procedural approach to working with data by handling the data row-by-row, set-based logic, or a set-based approach to handling SQL Server data is inherent to SQL Server. Using a set-based approach to querying data, you are able to manipulate the entire result set as a whole. When using set-based logic, you allow SQL Server to determine the most efficient way to retrieve and manipulate the result set.

Demonstration: Using Set-Based Queries

In this demonstration, you will learn how to:

- Use a set-based query to replace a cursor

Question: Why is it recommended to rewrite cursors as set-based queries?

Lesson 4

Dynamic SQL

Dynamic SQL

- Allows query to be built using variables
- Places query into variable

```
SET @SQLString = N'SELECT @SalesOrderOUT  
= MAX(SalesOrderNumber)  
FROM Sales.SalesOrderHeader  
WHERE CustomerID = @CustomerID';
```

Although static SQL works well in many situations, there are some applications in which the data access cannot be determined in advance. To solve this problem, an application can use a form of embedded SQL called dynamic SQL. Unlike static SQL statements, which are hard-coded in the program, dynamic SQL statements can be built at run time and placed in a string host variable. They are then sent to SQL Server for processing.

Introducing Dynamic SQL

Dynamic SQL

- Allows query to be built using variables
- Places query into variable

```
SET @SQLString = N'SELECT @SalesOrderOUT
= MAX(SalesOrderNumber)
FROM Sales.SalesOrderHeader
WHERE CustomerID = @CustomerID';
```

Key Points

Dynamic SQL allows a T-SQL query to be built using variables and strings, then executed as a string using two available methods, which you will learn about shortly.

Dynamic SQL can be useful in applications where forms with optional fields may be filled out and submitted. You can use the contents of the optional fields that have been filled out to create WHERE clauses to help return more relevant results.

In the code sample above, a variable called **@SQLString** is populated with a **SELECT** statement. The **@SQLString** variable can then be passed to one of two commands for execution.

Question: Can you think of a scenario where dynamic SQL might be useful?

Using Dynamic SQL

Using sp_executesql

```
sp_executesql [ @stmt = ] stmt  
[  
    {,[@params=] N'@parameter_name data_type [ OUT |  
OUTPUT ][,...n]' }  
    {,[@param1 = ] 'value1' [ ,...n ] }  
]
```

Using EXECUTE

```
[ { EXEC | EXECUTE } ]  
{ [ @return_status = ]  
{ module_name [ ;number ] | @module_name_var }  
[ [ @parameter = ] { value | @variable [ OUTPUT ]  
| [ DEFAULT ] } ] [ ,...n ] [ WITH RECOMPILE ] }  
[;]
```

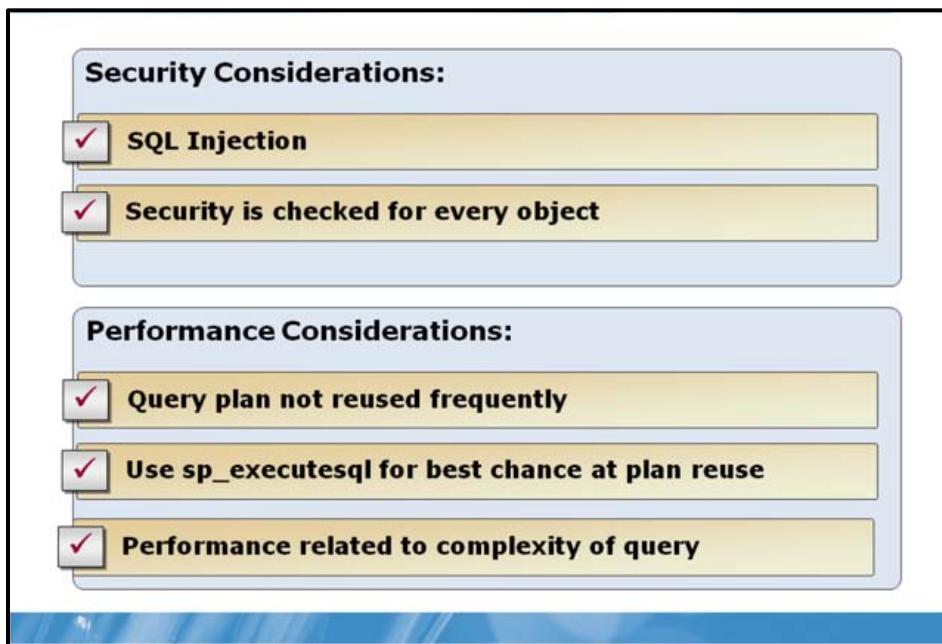
Key Points

There are two methods for executing dynamic SQL statements, **sp_executesql** and **EXECUTE**.

- **sp_executesql** is a system stored procedure that takes the dynamic SQL statement as a parameter and executes it. It can take additional parameters as well.
- **EXECUTE** can also execute dynamic SQL strings, and has a different syntax than **sp_executesql**.

Question: Why choose one method of dynamic SQL execution over the other?

Considerations for Using Dynamic SQL



Key Points

There are some important things to consider when using dynamic SQL:

- SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. Dynamic SQL can be prone to SQL injection attacks if strings are not validated before they are executed.
- Security is checked for every object involved in a dynamic SQL statement. This means that you may have to give more permissions on underlying database objects than you would with a stored procedure. A user with permission to execute a stored procedure does not need permissions on the underlying database objects, but a user who executed a dynamic SQL statement against those same tables must have permissions on them.
- Because of the nature of dynamic SQL, it is less likely to generate execution plans that can be cached and reused than static SQL. This can have an impact on query performance using dynamic SQL.
- Using **`sp_executesql`** can help SQL Server reuse execution plans when the only variation is in the parameter values supplied to the Transact-SQL statement. Because the Transact-SQL statements themselves remain constant and only the parameter values change, the SQL Server query optimizer is likely to reuse the execution plan it generates for the first execution.
- Because dynamic SQL statements are generally unlikely to generate reusable execution plans, their performance is related directly to the complexity of the query you build. When possible, try to reduce the complexity of any dynamic SQL queries you use.

Question: Why is SQL injection so dangerous?

Demonstration: Using Dynamic SQL

In this demonstration, you will learn how to:

- Build and execute a query that uses dynamic SQL

Question: What role do variables play in dynamic SQL?

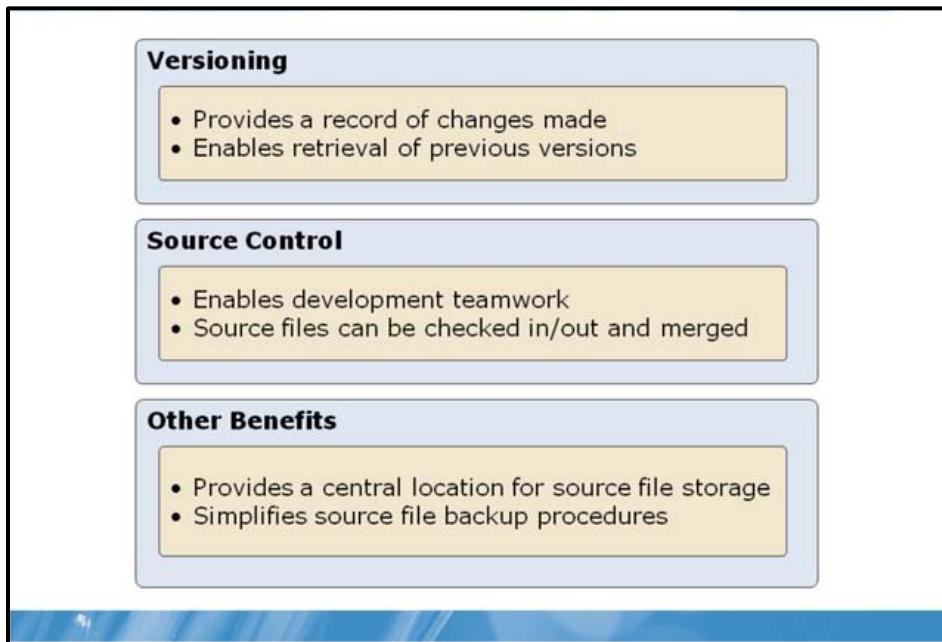
Lesson 5

Maintaining Query Files

- What Are Versioning and Source Control?
- Features of Team Foundation Server 2008

When you have many developers working on a SQL Server project, you want to have a historical record of versions of projects and files, or you want to ensure that you have a central repository for all your SQL Server projects, you may want to implement a versioning and source control system.

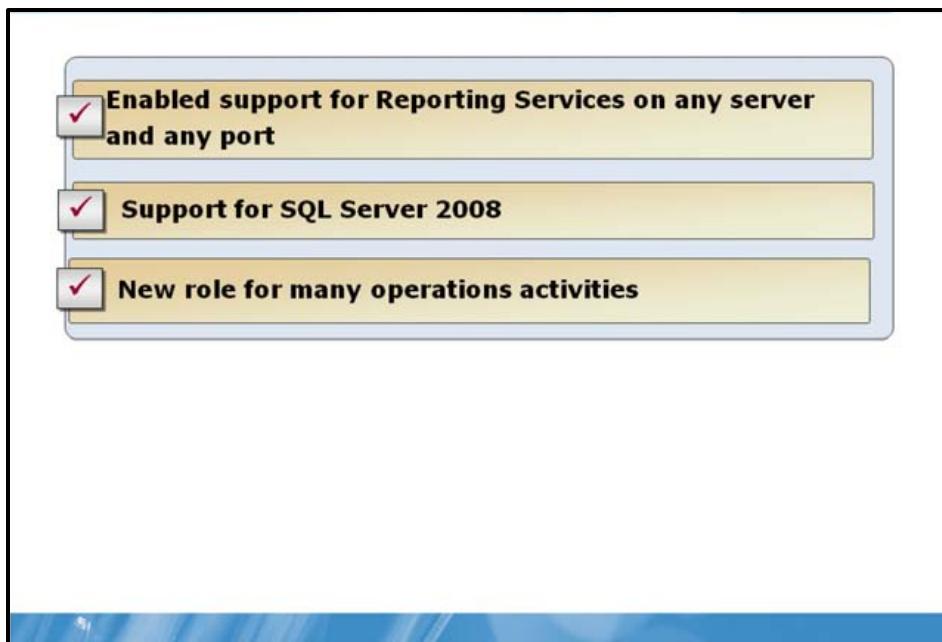
What Are Versioning and Source Control?



Key Points

When you work in teams, you need to enable the parallel development of projects by team members. However, when team members work with the same file, they need to reconcile conflicts between the different versions of the file and avoid the potential loss of valuable changes. In such situations, you require software such as Microsoft® Visual SourceSafe® or Team Foundation Server for source control and version management.

Features of Team Foundation Server 2008



Key Points

Team Foundation Server (TFS) 2008 provides source control and versioning capabilities. TFS will supports a Microsoft SQL Server 2005 back end, and will support SQL Server 2008 for a back end when SQL Server 2008 and Team Foundation Server SP1 reach RTM.

Lab 9: Using Advanced Techniques

- Using Execution Plans
- Converting Data Types
- Implementing a Hierarchy
- Using Cursors and Set-Based Queries

Logon information

Virtual machine	NY-SQL-01
User name	Administrator
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Exercise 1: Using Execution Plans

Scenario

You are a Database Administrator at Adventure Works. As part of an effort to analyze database performance, the Senior Database Administrator has asked you to become familiar with query execution plans. In order to do this, you need to know how to view estimated and actual execution plans, as well as save execution plans and view them in XML format.

The main tasks for this exercise are as follows:

1. Start the 2778A-NY-SQL-01 virtual machine and log on as Administrator.
 2. View an estimated execution plan.
 3. View an actual execution plan.
 4. Save an execution plan.
- **Task 1: Start the 2778A-NY-SQL-01 virtual machine and log on as Administrator**
- Start **NY-SQL-01** and log on as **Administrator** with a password of **Pa\$\$w0rd**.
- **Task 2: View an estimated execution plan**
- Use **SQL Server Management Studio** to open the file **E:\Mod09\Labfiles\Starter\Exercise01\Order-ProductID.sql**.
 - Display the **Estimated Execution Plan**.
 - Review the components of the execution plan.
- **Task 3: View an actual execution plan**
- Set the query to include the **Actual Execution Plan**.
 - Run the query.

- Review the components of the execution plan.

► **Task 4: Save an execution plan**

- Save the execution plan as a **.sqlplan** file to **E:\Mod09\Labfiles\Starter\Exercise01\Order-ProductID.sqlplan**.
- View the XML code for the execution plan and save it to **E:\Mod09\Labfiles\Starter\Exercise01\Order-ProductID.xml**.

Results: After this exercise, you should have viewed estimated and actual execution plans for queries, and saved the execution plans to files.

Exercise 2: Converting Data Types

Scenario

Some application queries have been producing unexpected results. The Senior Database Administrator suspects that some of the problems may be due to incorrect data type conversion. The Senior DBA has asked you to spend some time learning about data type conversion so that you can help resolve these problems.

The main tasks for this exercise are as follows:

1. Use implicit conversions.
2. Use explicit conversions.

► Task 1: Use implicit conversions

- Use **SQL Server Management Studio** to look up the data type for the **SalariedFlag** column in the **HumanResources.Employee** table in the AdventureWorks2008 database.
- Create a new query on the AdventureWorks2008 database with the following code:

```
SELECT NationalIDNumber, LoginID, SalariedFlag FROM HumanResources.Employee WHERE  
SalariedFlag = 'False'
```

- Review the **Results** pane. SQL Server automatically converted "**False**" to a data type of bit to compare it to the **SalariedFlag** column.
- Look up the data types for the **ListPrice** and **StandardCost** columns in the **Production.Product** table.
- Create a new query on the AdventureWorks2008 database with the following code:

```
DECLARE @NewListPrice nvarchar(10) = '39.99'  
DECLARE @NewStandardCost float = 12.45  
  
UPDATE Production.Product  
SET ListPrice = @NewListPrice, StandardCost = @NewStandardCost  
WHERE ProductID = '2'
```

- Run the query.
- Create a new query on the AdventureWorks2008 database with the following code:

```
SELECT Name, StandardCost, ListPrice from Production.Product WHERE ProductID = '2'
```

- Run the query.
- In the results pane, notice that the **ListPrice** and **StandardCost** columns have been updated. SQL Server automatically converted the **nvarchar** and **float** types in the update query to the **money** data type used by the columns.
- Open the file **E:\Mod09\Labfiles\Starter\Exercise02\UpdateDemographics.sql**.
- Notice that the variable **@NewDemographics** is an **nvarchar** data type and contains a valid XML string.
- Run the query.
- Create a new query on the AdventureWorks2008 database with the following code:

```
SELECT FirstName, MiddleName, LastName, Demographics FROM Person.Person WHERE  
BusinessEntityID = 15
```

- Run the query.
- In the Results pane, click the link in the **Demographics** column. SQL Server converted the **nvarchar** string to the **xml** data type automatically because the XML was valid.

► Task 2: Use explicit conversions

- Open the file **E:\Mod09\Labfiles\Starter\Exercise02\EmployeePayRate.sql**.
- Run the query and take note of the error message.
- Change the line of the query that begins with SELECT, so that it reads as follows:

```
SELECT (p.FirstName + ' ' + p.LastName + ' ' + CONVERT(nvarchar, s.Rate))
```

- Run the query. In the Results pane, there is a single column with the employee name and pay rate combined.
- Open the file **E:\Mod09\Labfiles\Starter\Exercise02\PayRateLastChange.sql**.
- Run the query.
- Review the results of the query. Notice that the last column does not show both the pay rate and date.
- In the query pane, modify the SELECT statement so that it begins as follows:

```
SELECT p.FirstName, p.LastName, (CAST(s.Rate AS nvarchar) + ' ' + s.RateChangeDate)
```

- Run the query, and take note of the error message.
- Modify the SELECT statement again, so that it reads as follows:

```
SELECT p.FirstName, p.LastName, (CAST(s.Rate AS nvarchar) + ' ' + CAST(s.RateChangeDate  
as nvarchar))
```

- Run the query. Notice that the query now runs and the last column displays the correct information.

Results: After this exercise, you should have verified that SQL Server automatically performs implicit conversions, and you should have also successfully run queries that performed explicit data type conversions.

Exercise 3: Implementing a Hierarchy

Scenario

The Sales department at Adventure Works is reorganizing its sales territories to reflect the organization's hierarchy. In order to represent the hierarchical structure of the sales territories, you have decided to create a new table in the database and implement the hierarchy using the hierarchyid data type.

The main tasks for this exercise are as follows:

1. Create a hierarchy.
2. Query a hierarchy.

► Task 1: Create a hierarchy

- Open the **Project E:\Mod09\Labfiles\Starter\Exercise03\SalesTerritory\SalesTerritory.ssmssln**.
- In **Solution Explorer**, open the **CreateTerritoryTable.sql** file.
- Review and then run the query.
- In **Solution Explorer**, open the **CreateTableIndex.sql** file.
- Review and then run the query.
- In **Solution Explorer**, open the **InsertRootTerritory.sql** file.
- Review and then run the query.
- In **Solution Explorer**, open the **InsertChildTerritory.sql** file.
- Review and then run the query.
- In **Solution Explorer**, open the **CreateTerritorySP.sql** file.
- Review and then run the query.
- In **Solution Explorer**, open the **PopulateTerritoryTable.sql** file.
- Review and then run the query.

► Task 2: Query a hierarchy

- In **Solution Explorer**, open the **GetTerritoryOrg.sql** file.
- Review and then run the query.
- Review the Results to see how the nodes in the hierarchy relate to one another. Pay particular attention to the **Text_OrgNode**, **OrgNode**, and **OrgLevel** columns.

Results: After this exercise, you should have created a table for the Sales department hierarchy and populated it.

Exercise 4: Using Cursors and Set-Based Queries

Scenario

You are working with a test copy of the AdventureWorks2008 database that has no pricing data in the **Production.Product** table. You need to populate both the **ListPrice** and **StandardCost** fields of the table, and have decided to use a cursor to update **ListPrice** data and use a set-based query to update **StandardCost** data to see how the two approaches compare.

The main tasks for this exercise are as follows:

1. Update the **ListPrice** column.
2. Update the **StandardCost** column.

► Task 1: Update the ListPrice column

- Create a new query on the AdventureWorks2008 database with the following code:

```
SELECT ProductID, Name, StandardCost, ListPrice FROM Production.Product
```

- Review the **StandardCost** and **ListPrice** columns in the Results pane. Keep this query pane open.
- Open the E:\Mod09\Labfiles\Starter\Exercise04\UpdateListPriceCursor.sql file.
- Review and execute the query.
- When the update is complete, run the SELECT query again.
- Review the **ListPrice** column in the Results pane. The **ListPrice** column has been updated.

► Task 2: Update the StandardCost column

- Create a new query on the AdventureWorks2008 database with the following code:

```
UPDATE Production.Product  
SET StandardCost = 7.99
```

- Run the query.
- When the update is complete, run the SELECT query again.
- Review the **StandardCost** column in the Results pane. The **StandardCost** column has been updated.
- Close **SQL Server Management Studio** without saving changes.
- Turn off the virtual machine and discard changes.

Results: After this exercise, you should have used a cursor to update the ListPrice column and used a set-based query to update the StandardCost column of the Production.Product table.

Module Review and Takeaways

- Review Questions
- Common Issues and Troubleshooting Tips
- Best Practices

Review Questions

1. How do cursors handle result sets?
2. What functions are used to perform explicit data type conversions?
3. What methods can be used to execute dynamic SQL queries?

Common Issues related to data type conversion

Identify the causes for the following common issues related to data type conversion and fill in the troubleshooting tips. For answers, refer to relevant lessons in the module.

Issue	Troubleshooting tip
Cannot implicitly or explicitly convert string or number data to xml data type	
Implicit conversions to binary data type produces unexpected results	
Converting integers to character data types implicitly results in an asterisk (*) instead of the correct value.	

Best Practices related to using dynamic SQL

Supplement or modify the following best practices for your own work situations:

- Because dynamic SQL is a significant vector of SQL injection attacks, there are a number of steps you should take. Make no assumptions about the size, type, or content of the data that is received by your application. For example, you should make the following evaluation:

- How will your application behave if an errant or malicious user enters a 10-megabyte MPEG file where your application expects a postal code?
- How will your application behave if a DROP TABLE statement is embedded in a text field.
- Test the size and data type of input and enforce appropriate limits. This can help prevent deliberate buffer overruns.
- Never build Transact-SQL statements directly from user input.
- Use stored procedures to validate user input.
- Never concatenate user input that is not validated. String concatenation is the primary point of entry for script injection.
- Do not accept the following strings in fields from which file names can be constructed: AUX, CLOCK\$, COM1 through COM8, CON, CONFIG\$, LPT1 through LPT8, NUL, and PRN.

Course Evaluation



Your evaluation of this course will help Microsoft understand the quality of your learning experience.

Please work with your training provider to access the course evaluation form.

Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.

Module 1: Getting Started with Databases and Transact-SQL in SQL Server 2008

Lab: Using SQL Server Management Studio and SQLCMD

Exercise 1: Explore the components and execute queries in SQL Server Management Studio

► Task 1: Launch the SQL Server Management Studio

1. In the Lab Launcher, next to **2778A-NY-SQL-01**, click **Launch**.
2. Log on to **NYC-SQL-01** as **Student** using the password **Pa\$\$w0rd**.
3. On the **Start** menu, click **All Programs**, click **Microsoft SQL Server 2008**, and then click **SQL Server Management Studio**.
4. The Microsoft SQL Server Management Studio window opens, and then the **Connect to Server** dialog box appears.
5. In the **Connect to Server** dialog box, verify that the **Server type**, **Server name**, and **Authentication** fields have been populated correctly, and then click **Connect** to accept the default settings.

► Task 2: Navigate through online help

1. On the **Help** menu, click **Contents**.
2. The Server 2008 Combined Help Collection - Microsoft Document Explorer window appears.
3. In the **Contents** pane, navigate to **SQL Server Books Online | Getting Started | Initial Installation | Overview of SQL Server Installation | Features and Tools Overview | SQL Server Studios Overview | Introducing SQL Server Management Studio | Tutorial: SQL Server Management Studio | Lesson 1: Basic Navigation in SQL Server Management Studio**.
4. The Lesson 1: Basic Navigation in SQL Server Management Studio page opens in the right pane.
5. In the **Contents** pane, expand **Lesson 1: Basic Navigation in SQL Server Management Studio**, and then click the topic **Connecting with Registered Servers and Object Explorer**.
6. The content of the page opens in the right pane.

► Task 3: Resize, hide, and close Object Explorer and Solution Explorer

1. Reduce the size of the **Contents** pane by resizing it.
2. Scroll down the Connecting with Registered Servers and Object Explorer page to view its contents.
3. Click the **Close** button to close the Help window.
4. On the **View** menu, click **Solution Explorer**.
5. In Solution Explorer, click **Auto Hide** to hide the pane.
6. In Object Explorer, click **Auto Hide** to hide the pane.

► Task 4: Create a new solution and explore the solution objects in Object Explorer

1. Open Object Explorer by clicking the **Object Explorer** tab.
2. In Object Explorer, click **Auto Hide** to dock the pane.
3. In the Object Explorer, double-click the **Databases** folder, double-click **AdventureWorks2008**, and then double-click **Tables**.
4. You can scroll through the pane to view the list of tables.
5. On the toolbar, click **New Query**. The SQLQuery1.sql - NY-SQL-01.AdventureWorks2008 window opens.

6. Open Solution Explorer by clicking the **Solution Explorer** tab.
7. In Solution Explorer, click **Auto Hide** to dock the pane.
8. In Solution Explorer, right-click **Solution 'Solution1' (1 project)**, point to **Add**, and then click **New Project**.
9. The **Add New Project** dialog box appears.
10. In the **Templates** box, verify that the **SQL Server Scripts** template is highlighted.
11. In the **Name** field, select **SQL Server Scripts1**, type **PersonAddress**, and then click **OK**.
12. The PersonAddress project will be created under Solution 'Solution1' (1 project).
13. In Solution Explorer, in the PersonAddress project, right-click **Connections**, and then click **New Connection**.
14. The **Connect to Server** dialog box appears. Verify the connection information, and then click **OK**.

► **Task 5: Add projects to an existing solution and create queries in the projects**

1. In the **Connections** folder of the PersonAddress project, right-click **NY-SQL-01:NY-SQL-01\Student**, and then click **New Query**.
2. The SQLQuery1.sql object appears in the **Queries** folder.
3. In the query window, type **USE AdventureWorks2008**, and then press ENTER.
4. On the toolbar, click **Execute**.
5. The active database is changed to **AdventureWorks2008**.
6. In the query window, type **SELECT DISTINCT CITY FROM Person.Address**.
7. On the toolbar, click **Execute**.
8. The list of cities in the **Person.Address** table is displayed without repetition in the **Results** pane.
9. In the **Queries** folder of PersonAddress project, right-click **SQLQuery1.sql**, and then click **Rename**.
10. Rename the query as **Address.sql**.
11. The query has been renamed to **Address.sql**.
12. On the **File** menu, click **Save Address.sql**.
13. In Solution Explorer, right-click **PersonAddress**, and then click **Save PersonAddress.ssmssqlproj**.
14. Right-click **Solution 'Solution1' (1 project)**, point to **Add**, and then click **New Project**.
15. The **Add New Project** dialog box appears.
16. In the **Templates** box, verify that the **SQL Server Scripts** template is highlighted.
17. In the **Name** field, select **SQL Server Scripts1**, type **HumanResourcesDepartment**, and then click **OK**.
18. The HumanResourcesDepartment project is created under Solution 'Solution1' (2 projects).
19. In the HumanResourceDepartments project, right-click **Connections**, and then click **New Connection**.
20. The **Connect to Server** dialog box appears.
21. Verify the connection information, and then click **OK**.
22. In the **Connections** folder, under **HumanResourceDepartments**, right-click **NY-SQL-01:NY-SQL-01\Student** and then click **New Query**.
23. The **SQLQuery1.sql** folder appears.
24. In the **Queries** folder of **HumanResourcesDepartment** project, right-click **SQLQuery1.sql**, and then click **Rename**.
25. Rename the query as **Department.sql**.
26. The query has been renamed to **Department.sql**.
27. In the query window, type **USE AdventureWorks2008**, and then press ENTER.
28. On the toolbar, click **Execute**.
29. The active database is changed to **AdventureWorks2008**.
30. In the query window, type **SELECT Name, GroupName FROM HumanResources.Department**.
31. On the toolbar, click **Execute**.

32. The **Name** and **GroupName** columns are displayed in the **Results** pane.
33. On the **File** menu, and click **Save Department.sql**.
34. In Solution Explorer, right-click the **HumanResourcesDepartment** project, and then click **Save HumanResourcesDepartment.ssmssqlproj**.
35. Click **Solution 'Solution1' (2 projects)**.
36. On the **File** menu, click **Save Solution1 As**.
37. The **Save File As** dialog box appears. Verify that the **Projects** folder is selected in the **Save in:** list.
38. In the **File name** field, type **AdventureWorks2008.ssmssqln**, and then click **Save**.
39. Click the **Close** button to close SQL Server Management Studio.

► **Task 6: Connect to SQL Server and execute a query**

1. On the **Start** menu, click **All Programs**, click **Microsoft SQL Server 2008**, and then click **SQL Server Management Studio**.
2. The Microsoft SQL Server Management Studio window opens, and then the **Connect to Server** dialog box appears.
3. In the **Connect to Server** dialog box, verify that the **Server type**, **Server name**, and **Authentication** fields have been populated correctly, and then click **Connect** to accept the default settings.
4. On the **File** menu, point to **Open**, and then click **File**.
5. The **Open File** dialog box appears. In the **Open File** window, navigate to the **PersonAddress** folder, click **Address.sql**, and then click **Open**.
6. If the Connect to Database Engine application window appears, click **Connect**.
7. On the toolbar, click **Execute**.
8. The results are displayed in the **Results** pane. It should be similar to the results in the earlier task.

► **Task 7: Use Visual Query Builder to return rows from a table**

1. In Object Explorer, expand the **Databases** folder.
2. Right-click **AdventureWorks2008** database, and then click **New Query**.
3. On the **Query** menu, click **Design Query in Editor**.
4. The **Query Designer** dialog box appears, and the **Add Table** dialog box appears.
5. In the **Add Table** dialog box, verify that the **Address (Person)** table is selected, and then click **Add**.
6. The **Address (Person)** dialog box appears.
7. The columns of the **Address** table are listed in the **Address (Person)** dialog box.
8. In the **Add Table** dialog box, click **Close**.
9. In the **Address (Person)** dialog box, select the *** (All Columns)** check box.
10. In the **Query Designer** dialog box, click **OK**.
11. Notice the query in the **SQLQuery1.sql - NY-SQL-01.AdventureWorks2008** query pane.
12. On the toolbar, click **Execute**.
13. All the columns in the **Person.Address** table are displayed in the **Results** pane.
14. Click the **Close** button to close SQL Server Management Studio.
15. The **Microsoft SQL Server Management Studio** dialog box appears.
16. Click **No** to close the solution without saving changes.

Results: After this exercise, you should have explored the components and executed queries in the SQL Server Management Studio.

Exercise 2: Start and use sqlcmd

- ▶ **Task 1: Start the sqlcmd utility and connect to a default instance of SQL Server**
 1. On the **Start** menu click **All Programs**, click **Accessories**, and then click **Command Prompt**.
 2. The Command Prompt window opens.
 3. At the command prompt, type **sqlcmd**, and then press ENTER.
 4. You now have a trusted connection to the default instance of SQL Server that is running on your computer. **1>** is the **sqlcmd** prompt that specifies the line number. Each time you press ENTER, the number increases by one.
 5. At the sqlcmd prompt, type **exit**, and then press ENTER, to end the sqlcmd session.
- ▶ **Task 2: Run a Transact-SQL script file by using sqlcmd**
 1. In the Command Prompt window, type: **sqlcmd -S NY-SQL-01 -i E:\MOD01\Labfiles\Starter\Department.sql**.
 2. Press ENTER.
 3. Notice that the results are similar to the results returned in the earlier task.
- ▶ **Task 3: Run a Transact-SQL script file and save the output to a text file**
 1. In the Command Prompt window, type: **sqlcmd -S NY-SQL-01 -i E:\MOD01\Labfiles\Starter\Department.sql -o E:\MOD01\Labfiles\Solution\DeptList.txt**.
 2. Press ENTER.
 3. Notice that there is no output in the Command Prompt window.
- ▶ **Task 4: Review the output file**
 1. On the **Start** menu, click **Computer**.
 2. The Windows Explorer window opens.
 3. Navigate to **E:\MOD01\Labfiles\Solution**.
 4. In the **details** pane, double-click **DeptList.txt**.
 5. The DeptList.txt - Notepad window opens.
 6. Notice that the results are similar to the results returned in the earlier task.
 7. Close Notepad.
 8. Close Windows Explorer.
 9. Close Command Prompt.

Results: After this exercise, you should have started and used sqlcmd to create reports.

Exercise 3: Generate a report from a SQL Server database using Microsoft Office Excel

► Task 1: Launch Excel

1. On the **Start** menu, click **All Programs**, click **Microsoft Office**, and then click **Microsoft Office Excel 2007**.
2. The Microsoft Excel window opens.

► Task 2: Create a new data connection in the workbook

1. On the **Data** menu, click **Get External Data**, click **From Other Sources**, and then click **From SQL Server**.
2. The **Data Connection Wizard** wizard appears.
3. In the **Server name** field, type **NY-SQL-01**.
4. Click **Next**.

► Task 3: Select the data to import and its format

1. On the Select Database and Table page, in the **Select the database that contains the data you want** list, click **AdventureWorks2008**.
2. In the **Connect to a specific table** box, in the **Name** column, click **Address**.
3. Click **Next**.
4. In the Save Data Connection File and Finish page, in the **Description** field, type **AdventureWorks2008 Addresses**.
5. In the **Friendly Name** field, type **Addresses**.
6. Click **Authentication Settings**.
7. The **Excel Services Authentication Settings** dialog box appears.
8. Notice the three options for authentication and notice that **Windows Authentication** is selected.
9. Click **Cancel**.
10. Click **Finish**.

► Task 4: View the report

1. The **Import Data** dialog box appears.
2. Notice the options for importing the data.
3. Click **OK** to accept the default settings.
4. The **Microsoft Excel-Book 1** is populated with the names of various cities.
5. Examine the results.
6. Close Microsoft Office Excel.
7. The **Microsoft Office Excel** dialog box appears indicating whether you need to save the changes that you have made to **Book 1**.
8. Click **No**.
9. Turn off 2778A-NY-SQL-01 and delete changes.

Results: After this exercise, you should create a report from a SQL Server database using Microsoft Office Excel.

Module 2: Querying and Filtering Data

Lab: Querying and Filtering Data

Exercise 1: Exploring SQL Server Management Studio

► **Task 1: Launch SQL Server Management Studio**

1. In the Lab Launcher, next to **2778A-NY-SQL-01**, click **Launch**.
2. Log on as **Administrator** with the password of **Pa\$\$w0rd**.

► **Task 2: Generate a report by using the SELECT * statement with the FROM clause**

1. On the **Start** menu, point to **All Programs**, click **Microsoft SQL Server 2008** and click **SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, verify that the **Server type**, **Server name**, and the **Authentication** boxes have been populated correctly, and click **Connect** to accept the default settings.
3. On the toolbar of SQL Server Management Studio, click **New Query**. A new query opens in the query window.
4. In the query window type the following SQL Statement:

```
USE AdventureWorks2008  
GO  
SELECT *  
FROM Person.Person
```

5. Click **Execute**.
6. Browse the result set in the **Results** pane.

► **Task 3: Generate a report by using the SELECT statement with the FROM clause**

1. On the toolbar of SQL Server Management Studio, click **New Query**. A new query opens in the query window.
2. In the new query window type and execute the following SQL statement:

```
USE AdventureWorks2008  
GO  
SELECT Title, FirstName, LastName, MiddleName  
FROM Person.Person
```

3. Click **Execute**. The result set is displayed in the **Results** pane.
4. Browse the result set in the **Results** pane and notice that only the specified columns appear.

Results: After this exercise, you should have learned how to create a basic SELECT statement to retrieve data from a table.

Exercise 2: Filter Data by Using Various Search Conditions

► **Task 1: Generate a report by using the SELECT statement with a COMPARISON operator**

1. On the toolbar of **SQL Server Management Studio**, click **New Query**. A new query opens in the query window.
2. In the query window enter and execute the following SQL statement:

```
USE AdventureWorks2008
GO
SELECT ProductNumber, Name, ListPrice, Color, Size, Weight
FROM Production.Product
WHERE ListPrice < $100
```

3. Click **Execute**. The result set is displayed in the **Results** pane.

► **Task 2: Generate a report by using the SELECT statement with the AND and LIKE operators**

1. Enter the following SQL Statement in the Query window:

```
USE AdventureWorks2008
GO
SELECT ProductNumber, Name, ListPrice, Color, Size, Weight
FROM Production.Product
WHERE ListPrice < $100
AND ProductNumber LIKE 'S0%'
```

2. Click **Execute**. The result set is displayed in the **Results** pane.

► **Task 3: Generate a report by using the SELECT statement with the OR operator**

1. Change the SQL statement to read:

```
USE AdventureWorks2008
GO
SELECT ProductNumber, Name, ListPrice, Color, Size, Weight
FROM Production.Product
WHERE ListPrice < $100
AND ProductNumber Like 'S0%'
OR ProductNumber Like 'TG%'
```

2. Click **Execute**. The result set is displayed in the **Results** pane.

► **Task 4: Generate a report by using the SELECT statement with the BETWEEN operator**

1. Click **New Query** and type the following SQL statement into the Query window:

```
USE AdventureWorks2008
GO
SELECT ProductNumber, Name, ListPrice, Color, Size, Weight
FROM Production.Product
WHERE ProductNumber LIKE 'S0%'
OR (ListPrice Between $50 and $180
    AND ProductNumber LIKE 'TG%')
```

2. Click **Execute**. The result set is displayed in the **Results** pane.

► **Task 5: Generate a report by using the SELECT statement with the IN operator**

1. Rewrite the SQL statement to read as follows:

```
USE AdventureWorks2008
GO
SELECT ProductNumber, Name, ListPrice, Color, Size, Weight
FROM Production.Product
WHERE ProductNumber LIKE 'S0%'
OR (ListPrice Between $50 and $180
    AND ProductNumber LIKE 'TG%')
```

```
AND Size IN ('M', 'L'))
```

2. Click **Execute**. The result set is displayed in the **Results** pane.

Results: After this exercise, you should have learned how to use several different comparison operators to create reports specific to different user needs.

Exercise 3: Using Functions to Work with Null Values

► **Task 1: Generate a report by using the SELECT statement with the NULL function**

1. On the toolbar of SQL Server Management Studio, click **New Query**. A new query opens in the query window.
2. In the query window, type the following SQL statements:

```
USE AdventureWorks2008
GO
SELECT ProductNumber, Name, Weight
FROM Production.Product
WHERE ProductLine = NULL
```

3. Click **Execute**.
4. Notice that no result set is displayed in the **Results** pane.

► **Task 2: Generate a report by using the SELECT statement with the IS NULL function**

1. Rewrite the SQL statement to use **IS NULL**:

```
USE AdventureWorks2008
GO
SELECT ProductNumber, Name, Weight
FROM Production.Product
WHERE ProductLine IS NULL
```

2. Click **Execute**.
3. Browse the result set displayed in the **Results** pane.

► **Task 3: Generate a report by using the SELECT statement with the ISNULL function to rename values**

1. On the toolbar of SQL Server Management Studio, click **New Query**. A new query opens in the query window.
2. In the query window type the following SQL statement:

```
USE AdventureWorks2008
GO
SELECT ProductNumber, Name, Weight,
ISNULL(ProductLine, 'NA')
FROM Production.Product
```

3. Click **Execute**.
4. The result set is displayed in the **Results** pane. Notice that the new column shows **NA** in all rows that Weight has a **NULL** value. But the column has no heading.

► **Task 4: Generate a report by using the SELECT statement with the ISNULL function and a column alias**

1. Change the statement as follows, to add **Product Line** as a column heading for the **ProductLine** column:

```
USE AdventureWorks2008
GO
SELECT ProductNumber, Name, Weight,
ISNULL(ProductLine, 'NA') AS 'Product Line'
```

```
FROM Production.Product
```

2. Click **Execute**.
3. The result set is displayed in the **Results** pane. Note that the new column now has the heading of **Product Line**.

► **Task 5: Generate a report by using the SELECT statement with the ISNULL function and the COALESCE and CONVERT functions**

1. Click **New Query** and enter the following SQL Statement into the Query window:

```
USE AdventureWorks2008
GO
SELECT ProductNumber, Name, COALESCE(CONVERT(NVARCHAR,Weight), SIZE, 'NA') AS Measurement,
ISNULL(ProductLine, 'NA') AS 'Product Line'
FROM Production.Product
```

2. Click **Execute**.
3. The result set is displayed in the **Results** pane. Notice the values in the **Measurement** column.

Results: After this exercise, you should have learned to handle NULL values in a result set by identifying them and replacing them with alternate values when necessary.

Exercise 4: Formatting Result Sets

► **Task 1: Format a result set by using the ORDER BY clause**

1. Click **New Query**.
2. In the new query window type the following SQL statement:

```
USE AdventureWorks2008
GO
SELECT ProductNumber, Name, Class
FROM Production.Product
ORDER BY Class
```

3. Click **Execute**.

► **Task 2: Format a result set by using the ORDER BY clause and the DESC keyword**

1. Click **New Query**.
2. Type the following SQL statement into the Query window:

```
USE AdventureWorks2008
GO
SELECT ProductNumber, Name, Class, ListPrice
FROM Production.Product
ORDER BY Class, ListPrice Desc
```

3. Click **Execute**. The result set is displayed in the **Results** pane.

► **Task 3: Format a result set by using the DISTINCT keyword**

1. Click **New Query**.
2. In the query window, enter the following SQL statement:

```
USE AdventureWorks2008
```

```
GO  
SELECT DISTINCT Color  
FROM Production.Product  
WHERE ProductNumber LIKE 'HL%'
```

3. Click **Execute**. The result set is displayed in the **Results** pane.

► **Task 4: Format a result set by concatenating strings**

1. Click **New Query**.
2. In the query window type the following SQL statement:

```
USE AdventureWorks2008  
GO  
SELECT LastName + ',' + FirstName  
FROM Person.Person
```

3. Click **Execute**. The result set is displayed in the **Results** pane.

► **Task 5: Format a result set by concatenating strings and using column aliasing**

1. Rewrite the query to assign the column name **Contacts**.

```
USE AdventureWorks2008  
GO  
SELECT (LastName + ',' + FirstName)  
AS Contacts  
FROM Person.Person
```

2. Click **Execute**. The result set is displayed in the **Results** pane.

► **Task 6: Format a result set by using the SUBSTRING function**

1. Rewrite the statement so that it will search for all rows with the **LastName** beginning with **Mac**.

```
USE AdventureWorks2008  
GO  
SELECT (LastName + ',' + FirstName) AS Contacts  
FROM Person.Person  
WHERE SUBSTRING (LastName,1,3)= 'Mac'
```

2. Click **Execute**. The result set is displayed in the **Results** pane.

3. On the **File** menu, click **Exit** and answer **No** to the save queries prompt.

4. Turn off NY-SQL-01 and discard changes.

Results: After this exercise, you should have learned how to format the result sets to make them more readable.

Module 3: Grouping and Summarizing Data

Lab: Grouping and Summarizing Data

Exercise 1: Summarizing Data by Using Aggregate Functions

► **Task 1: Start the 2778A-NY-SQL-01 virtual machine, log on as Administrator, and launch SQL Server Management Studio**

1. In the Lab Launcher, next to **2778A-NY-SQL-01**, click **Launch**.
2. Log on as **Administrator** with the password of **Pa\$\$w0rd**.
3. Click **Start | All Programs | Microsoft SQL Server 2008**, and then click **SQL Server Management Studio**.
4. In the **Connect to Server** dialog box, verify that **Server type** is set to **Database Engine** and **Server name** is set to **NY-SQL-01**, and then click **Connect**.
5. Maximize Microsoft SQL Server Management Studio.
6. In Object Explorer, expand **NY-SQL-01 | Databases**.

► **Task 2: Write a query that displays a single summary value for all the rows**

1. On the toolbar of SQL Server Management Studio, click **New Query**. A new query opens in the query window.
2. In the **Available Databases** list on the toolbar, click **AdventureWorks2008**. You are connected to the **AdventureWorks2008** database.
3. In the query window, type the following SQL statements:

```
SELECT AVG(VacationHours)
AS 'AverageVacationHours',
SUM(SickLeaveHours) AS 'TotalSickLeave Hours'
FROM HumanResources.Employee
WHERE JobTitle LIKE '%Vice President%'
```

4. On the toolbar, click **Execute**.

► **Task 3: Write a second query that displays a single summary value for all rows**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statement:

```
SELECT COUNT(*)
FROM HumanResources.Employee
```

3. Select the statement, and on the toolbar click **Execute**.

► **Task 4: Write a query that computes the total number of employees with the AddressLine2 value as NULL**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statement:

```
SELECT COUNT(*)
FROM Person.Address
WHERE ISNULL (AddressLine2, '0') = '0'
```

3. Select the statement, and on the toolbar click **Execute**.

► **Task 5: Modify the query and eliminate the NULL values**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statement:

```
SELECT COUNT (AddressLine2)
FROM Person.Address
```

3. Select the query, and on the toolbar, click **Execute**.
4. On the toolbar of SQL Server Management Studio, click **New Query**.
5. In the query window, type the SQL statement:

```
SELECT COUNT (DISTINCT AddressLine2)
FROM Person.Address
```

6. Select the statement, and on the toolbar click **Execute**.

Results: After this exercise you should have launched SQL Server Management Studio and created queries to display summary values for rows. You should have also created queries to compute totals and eliminate NULL values.

Exercise 2: Summarizing Grouped Data

► **Task 1: Write a query that computes the average number of days to manufacture a product**

1. On the toolbar of SQL Server Management Studio, click **New Query**. A new query opens in the query window.
2. In the **Available Databases** list on the toolbar, click **AdventureWorks2008**. You are connected to the **AdventureWorks2008** database.
3. In the query window, type the following SQL statements:

```
SELECT ProductID, AVG(DaysToManufacture)
AS 'AvgDaysToManufacture'
FROM Production.Product
GROUP BY ALL ProductID
```

4. On the toolbar, click **Execute**.

► **Task 2: Write a query that displays the various colors of a particular product and the average ListPrice of the colors**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statements:

```
SELECT Color, AVG(ListPrice) AS 'AvgListPrice'
FROM Production.Product
WHERE ProductNumber = 'FR-R72R-58'
GROUP BY ALL Color
```

3. Select the statement, and on the toolbar click **Execute**.

► **Task 3: Generate a report that lists the average order quantity and sum of line total for each product with a line total that exceeds \$1000000.00 and with average quantity less than 3**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statements:

```
SELECT ProductID, AVG(OrderQty)
AS 'AverageQuantity', SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
GROUP BY ProductID
HAVING SUM(LineTotal) > $1000000.00 AND AVG(OrderQty) < 3
```

3. Select the statement, and on the toolbar click **Execute**.

► **Task 4: Group the products and then compute the sum of the quantity shelf-wise**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statements:

```
SELECT ProductID, Shelf, SUM(Quantity) AS 'QtySum'
FROM Production.ProductInventory
GROUP BY ROLLUP(ProductID, Shelf)
```

3. Select the statement, and on the toolbar click **Execute**.

► **Task 5: Generate a summary report**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statements:

```
SELECT SalesOrderID, ProductID,
SUM(OrderQty) AS SumQuantity
FROM Sales.SalesOrderDetail
GROUP BY CUBE(SalesOrderID, ProductID)
ORDER BY SalesOrderID, ProductID
```

3. Select the statement, and on the toolbar click **Execute**.

► **Task 6: Distinguish the rows generated by the summary or aggregations and actual table rows**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statements:

```
SELECT SalesQuota, SUM(SalesYTD) 'TotalSalesYTD',
GROUPING(SalesQuota) AS 'Grouping'
FROM sales.SalesPerson
GROUP BY CUBE(SalesQuota)
```

3. Select the statement, and on the toolbar click **Execute**.

► **Task 7: Generate a report of the summary columns by using the GROUPING function**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statements:

```
SELECT ProductID, Shelf, SUM(Quantity) AS 'TotalQuantity',
GROUPING (Shelf) AS 'Shelfgrouping'
FROM Production.ProductInventory
GROUP BY CUBE(ProductID, Shelf)
```

3. Select the statement, and on the toolbar click **Execute**.

► **Task 8: Generate a report that displays all the products, the unit price, the unit price discount, and the sum of the columns**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statements:

```
SELECT SalesOrderID, UnitPrice, UnitPriceDiscount
FROM Sales.SalesOrderDetail
ORDER BY SalesOrderID
COMPUTE SUM(UnitPrice), SUM(UnitPriceDiscount)
```

3. Select the statement, and on the toolbar click **Execute**.

Results: After this exercise you have learned how to create queries to compute averages, display additional data and perform summations. You should have also learned how to create queries using aggregation functions and the GROUPING function.

Exercise 3: Ranking Grouped Data

► **Task 1: Generate row numbers for each of the rows in the year-to-date sales of the salespersons**

1. On the toolbar of SQL Server Management Studio, click **New Query**. A new query opens in the query window.
2. In the **Available Databases** list on the toolbar, click **AdventureWorks2008**. You are connected to the **AdventureWorks2008** database.
3. In the query window, type the following SQL statements:

```
SELECT ROW_NUMBER() OVER(ORDER BY SalesYTD DESC) AS 'Row Number', BusinessEntityID,
SalesYTD
FROM Sales.SalesPerson S
WHERE TerritoryID IS NOT NULL AND SalesYTD<>0
```

4. On the toolbar, click **Execute**.

► **Task 2: Generate a report that ranks the products by using the RANK function**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statements:

```
SELECT RANK() OVER (PARTITION BY LocationID ORDER BY Quantity DESC) AS RANK,
ProductID, LocationID, Quantity
FROM Production.ProductInventory
ORDER BY RANK
```

3. Select the statement, and on the toolbar click **Execute**.

► **Task 3: Generate a report that ranks the products by using the DENSE_RANK function**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statements:

```
SELECT DENSE_RANK() OVER (PARTITION BY LocationID ORDER BY Quantity DESC) AS RANK,
ProductID, LocationID, Quantity
FROM Production.ProductInventory
ORDER BY RANK
```

3. Select the statement, and on the toolbar click **Execute**.

► **Task 4: Generate a report that lists the details of salespersons based on their year-to-date sales in descending order, and then group them into four categories**

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statements:

```
SELECT NTILE(4) OVER(ORDER BY SalesYTD DESC) AS 'Quartile', SalesYTD, BusinessEntityID
FROM Sales.SalesPerson
WHERE TerritoryID IS NOT NULL AND SalesYTD <> 0
```

3. Select the statement and on the toolbar, click **Execute**.

Results: After this exercise you have learned how to create queries that use the various ranking functions to organize and rank grouped data.

Exercise 4: Creating Crosstab Queries

► Task 1: Execute a simple SELECT statement using the PIVOT operator

1. On the toolbar of SQL Server Management Studio, click **New Query**. A new query opens in the query window.
2. In the **Available Databases** list on the toolbar, click **AdventureWorks2008**. You are connected to the **AdventureWorks2008** database.
3. In the query window, type the following SQL statements:

```
SELECT Name, [RED], [BLUE], [BLACK]
FROM (SELECT SafetyStockLevel, Color, Name FROM
Production.Product) P
PIVOT
(
    SUM (SafetyStockLevel) FOR Color IN ([RED], [BLUE], [BLACK])
) AS PVT
WHERE Name LIKE '%Helmet%' ORDER BY Name
```

4. On the toolbar, click **Execute**.

► Task 2: Execute a simple SELECT statement by using the UNPIVOT operator

1. On the toolbar of SQL Server Management Studio, click **New Query**.
2. In the query window, type the following SQL statements:

```
SELECT Name, Attribute, Value
FROM
(SELECT Name, CAST (ProductLine as SQL_Variant) ProductLine,
CAST (StandardCost as Sql_variant) StandardCost,
CAST (ListPrice as sql_variant) ListPrice
FROM Production.Product) P UNPIVOT (Value FOR Attribute
IN ([ProductLine], [StandardCost], [ListPrice]))
AS UnPVT Order By Name Desc
```

3. Select the statement, and on the toolbar click **Execute**.
4. Turn off the 2778A-NY-SQL-01 virtual machine and discard any changes.

Results: After this exercise you have learned how to use the PIVOT operator to create a crosstab query and also how to use the UNPIVOT operator.

Module 4: Joining Data from Multiple Table

Lab: Joining Data from Multiple Tables

Exercise 1: Querying Multiple Tables by Using Joins

► Task 1: Launch SQL Server Management Studio

1. In the Lab Launcher, next to **2778A-NY-SQL-01**, click **Launch**.
2. Log on as **Administrator** with the password of **Pa\$\$w0rd**.
3. Start Windows Explorer and browse to **E:\MOD04\Labfiles\Starter**.
4. Double-click **LabSetup.cmd**.
5. Close Windows Explorer.
6. Click **Start | All Programs | Microsoft SQL Server 2008**, and then click **SQL Server Management Studio**.
7. In the **Connect to Server** dialog box, verify that **Server type** is set to **Database Engine** and **Server name** is set to **NY-SQL-01**, and then click **Connect**.
8. Maximize Microsoft SQL Server Management Studio.
9. In Object Explorer, expand **NY-SQL-01 | Databases**.

► Task 2: Create and execute an inner join

1. Right-click the **AdventureWorks2008** database and then click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT e.LoginID  
FROM HumanResources.Employee AS e  
INNER JOIN Sales.SalesPerson AS s  
ON e.BusinessEntityID = s.BusinessEntityID;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 3: Create and execute a left outer join and a right outer join

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT p.Name, pr.ProductReviewID  
FROM Production.Product p  
LEFT OUTER JOIN Production.ProductReview pr  
ON p.ProductID = pr.ProductID;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. On the toolbar, click **New Query**.
6. In the new, blank query window type the following T-SQL statement:

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

7. On the toolbar, click **Execute**.

8. Review the results of the query and then close the query editor without saving the query.
9. Leave Microsoft SQL Server Management Studio open for the next exercise.

Results: After this exercise, you should have launched SQL Server Management Studio and created and executed an Inner Join. You should have also created and executed a Left Outer Join and a Right Outer Join.

Exercise 2: Applying Joins for Typical Reporting Needs

► Task 1: Create and execute a query using a self join

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT pv1.ProductID, pv1.BusinessEntityID
FROM Purchasing.ProductVendor pv1
INNER JOIN Purchasing.ProductVendor pv2
ON pv1.ProductID = pv2.ProductID
AND pv1.BusinessEntityID <> pv2.BusinessEntityID
ORDER BY pv1.ProductID;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 2: Create and execute a query using a self join and the DISTINCT clause

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT DISTINCT pv1.ProductID, pv1.BusinessEntityID
FROM Purchasing.ProductVendor pv1
INNER JOIN Purchasing.ProductVendor pv2
ON pv1.ProductID = pv2.ProductID
AND pv1.BusinessEntityID <> pv2.BusinessEntityID
ORDER BY pv1.ProductID;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 3: Create and execute a non-equi join with an equality and a non-equality operator

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT DISTINCT p1.ProductSubcategoryID, p1.ListPrice
FROM Production.Product p1
INNER JOIN Production.Product p2
ON p1.ProductSubcategoryID = p2.ProductSubcategoryID
AND p1.ListPrice <> p2.ListPrice
WHERE p1.ListPrice < $15 AND p2.ListPrice < $15
ORDER BY ProductSubcategoryID;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Leave Microsoft SQL Server Management Studio open for the next exercise.

Results: After this exercise, you should have created and executed a query using a self join, created and executed a query using a self join and the DISTINCT clause, and created and executed a non-equi join with both an equality and non-equality operator.

Exercise 3: Combining and Limiting Result Sets

► **Task 1: Combine the result sets of two queries by using the UNION ALL operator**

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT * FROM TestA  
UNION ALL  
SELECT * FROM TestB;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► **Task 2: Limit result sets by using the EXCEPT clause with the SELECT statement**

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT ProductID  
FROM Production.Product  
EXCEPT  
SELECT ProductID  
FROM Production.WorkOrder;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► **Task 3: Limit result sets by using the INTERSECT clause with the SELECT statement**

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT ProductID  
FROM Production.Product  
INTERSECT  
SELECT ProductID  
FROM Production.WorkOrder;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► **Task 4: Limit result sets by using the TOP and TABLESAMPLE operators**

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT TOP(15) PERCENT  
ProductID  
FROM Production.Product  
ORDER BY ProductID;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. On the toolbar, click **New Query**.
6. In the new, blank query window type the following T-SQL statement:

```
SELECT FirstName, LastName  
FROM Person.Person  
TABLESAMPLE (10 PERCENT);
```

7. On the toolbar, click **Execute**.
8. Review the results of the query and then close the query editor without saving the query.
9. Turn off the 2778A-NY-SQL-01 virtual machine and discard any changes.

Results: After this exercise, you should have combined the result sets of two queries by using the UNION ALL operator. You should have also limited result sets by using the EXCEPT and INTERSECT clauses. Finally you should have limited result sets using the TOP and TABLESAMPLE operators.

Module 5: Working with Subqueries

Lab: Working with Subqueries

Exercise 1: Writing Basic Subqueries

► Task 1: Launch SQL Server Management Studio

1. In the Lab Launcher, next to **2778A-NY-SQL-01**, click **Launch**.
2. Log on as **Administrator** with the password of **Pa\$\$w0rd**.
3. Click **Start | All Programs | Microsoft SQL Server 2008**, and then click **SQL Server Management Studio**.
4. In the **Connect to Server** dialog box, verify that **Server type** is set to **Database Engine** and **Server name** is set to **NY-SQL-01**, and then click **Connect**.
5. Maximize **Microsoft SQL Server Management Studio**.
6. In Object Explorer ensure that **NY-SQL-01 (SQL Server 10.0.1600 - NY-SQL-01\Administrator)** is expanded and then expand **Databases**.

► Task 2: Create a basic subquery

1. Right-click the **AdventureWorks2008** database and then click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT Name
FROM Production.Product
WHERE ListPrice =
    (SELECT ListPrice
     FROM Production.Product
     WHERE Name = 'Chainring Bolts');
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 3: Create a subquery with the EXISTS keyword

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT Name
FROM Production.Product
WHERE EXISTS
    (SELECT *
     FROM Production.ProductSubcategory
     WHERE ProductSubcategoryID =
           Production.Product.ProductSubcategoryID
     AND Name = 'Wheels');
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Leave **Microsoft SQL Server Management Studio** open for the next exercise.

Results: After this exercise, you should have launched SQL Server Management Studio and created and executed a basic subquery. You should have also created and executed a subquery with the EXISTS keyword.

Exercise 2: Writing Correlated Subqueries

► Task 1: Create a correlated subquery

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT DISTINCT c.LastName, c.FirstName  
FROM Person.Person c JOIN HumanResources.Employee e  
ON e.BusinessEntityID = c.BusinessEntityID  
WHERE 5000.00 IN  
(SELECT Bonus  
FROM Sales.SalesPerson sp  
WHERE e.BusinessEntityID = sp.BusinessEntityID);
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 2: Create a correlated subquery with comparison operators

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT ProductID, OrderQty  
FROM Sales.SalesOrderDetail s1  
WHERE s1.OrderQty <  
(SELECT AVG (s2.OrderQty)  
FROM Sales.SalesOrderDetail s2  
WHERE s2.ProductID = s1.ProductID);
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Leave **Microsoft SQL Server Management Studio** open for the next exercise.

Results: After this exercise, you should have created a correlated subquery and also created a correlated subquery with comparison operators.

Exercise 3: Comparing Subqueries with Joins and Temporary Tables

► Task 1: Create a subquery and a join that produce the same result set

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

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

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

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 2: Create a temporary table

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
CREATE TABLE #MyTempTable (cola INT PRIMARY KEY);
INSERT INTO #MyTempTable VALUES (1);
SELECT * FROM #MyTempTable;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Leave **Microsoft SQL Server Management Studio** open for the next exercise.

Results: After this exercise you should have created a subquery and a join that produced the same result set. You should have also created a temporary table.

Exercise 4: Using Common Table Expressions

► **Task 1: Create a common table expression (CTE)**

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
WITH Sales_CTE (SalesPersonID, NumberOfOrders, MaxDate)
AS
(
    SELECT SalesPersonID, COUNT(*) , MAX(OrderDate)
    FROM Sales.SalesOrderHeader
    GROUP BY SalesPersonID
)
SELECT E.BusinessEntityID, OS.NumberOfOrders, OS.MaxDate
FROM HumanResources.Employee AS E
JOIN Sales_CTE AS OS
    ON E.BusinessEntityID = OS.SalesPersonID
ORDER BY E.BusinessEntityID;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► **Task 2: Create a recursive query using a CTE**

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
USE AdventureWorks2008;
GO
WITH Parts(AssemblyID, ComponentID, PerAssemblyQty, EndDate, ComponentLevel) AS
(
    SELECT b.ProductAssemblyID, b.ComponentID, b.PerAssemblyQty,
           b.EndDate, 0 AS ComponentLevel
    FROM Production.BillOfMaterials AS b
    WHERE b.ProductAssemblyID = 800
          AND b.EndDate IS NULL
    UNION ALL
    SELECT bom.ProductAssemblyID, bom.ComponentID, p.PerAssemblyQty,
           bom.EndDate, ComponentLevel + 1
    FROM Production.BillOfMaterials AS bom
    INNER JOIN Parts AS p
        ON bom.ProductAssemblyID = p.ComponentID
        AND bom.EndDate IS NULL
)
SELECT AssemblyID, ComponentID, Name, PerAssemblyQty, EndDate,
       ComponentLevel
FROM Parts AS p
INNER JOIN Production.Product AS pr
    ON p.ComponentID = pr.ProductID
ORDER BY ComponentLevel, AssemblyID, ComponentID;
GO
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Turn off the 2778A-NY-SQL-01 virtual machine and discard any changes.

Results: After this exercise you should have created a common table expression. You should have also created a recursive query using a common table expression.

Module 6: Modifying Data in Tables

Lab: Modifying Data

Exercise 1: Inserting Data into Tables

► Task 1: Launch SQL Server Management Studio

1. In the Lab Launcher, next to **2778A-NY-SQL-01**, click **Launch**.
2. Log on as **Administrator** with the password of **Pa\$\$w0rd**.
3. Click **Start | All Programs | Microsoft SQL Server 2008**, and then click **SQL Server Management Studio**.
4. In the **Connect to Server** dialog box, verify that **Server type** is set to **Database Engine** and **Server name** is set to **NY-SQL-01**, and then click **Connect**.
5. Maximize **Microsoft SQL Server Management Studio**.
6. In Object Explorer expand **NY-SQL-01 | Databases**.

► Task 2: Create an INSERT statement that adds values to a table

1. Right-click the AdventureWorks2008 database and then click New Query.
2. In the new, blank query window type the following T-SQL statement:

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

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 3: Create an INSERT statement that adds multiple rows to a table

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
INSERT INTO Person.PersonPhone  
VALUES (N'1705', N'864-555-2101', N'3', GETDATE()),  
(N'1706', N'712-555-0118', N'1', GETDATE());
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 4: Create an INSERT statement using the OUTPUT syntax

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
DECLARE @MyTableVar table( ScrapReasonID smallint,  
                           Name varchar(50),  
                           ModifiedDate datetime);  
  
INSERT Production.ScrapReason  
      OUTPUT INSERTED.ScrapReasonID, INSERTED.Name, INSERTED.ModifiedDate  
        INTO @MyTableVar  
      VALUES (N'Operator error', GETDATE());  
  
--Display the result set of the table variable.  
SELECT ScrapReasonID, Name, ModifiedDate FROM @MyTableVar;  
--Display the result set of the table.  
SELECT ScrapReasonID, Name, ModifiedDate
```

```
FROM Production.ScrapReason;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Leave **Microsoft SQL Server Management Studio** open for the next exercise.

Results: After this exercise, you should have launched **SQL Server Management Studio**, created and executed an INSERT statement that adds values to a table, created and executed an INSERT statement using the INTO syntax, and created and executed an INSERT statement using the OUTPUT syntax.

Exercise 2: Deleting Data from Tables

► Task 1: Create a DELETE statement using the WHERE syntax

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
DELETE FROM Production.ProductCostHistory  
WHERE StandardCost > 1000.00;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 2: Create a DELETE statement using the FROM syntax

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
DELETE FROM Sales.SalesPersonQuotaHistory  
FROM Sales.SalesPersonQuotaHistory AS sqqh  
    INNER JOIN Sales.SalesPerson AS sp  
        ON sqqh.BusinessEntityID = sp.BusinessEntityID  
WHERE sp.SalesYTD > 1000000.00;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 3: Create a DELETE statement using the OUTPUT syntax

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
DELETE Sales.ShoppingCartItem  
    OUTPUT DELETED.*;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 4: Create a TRUNCATE TABLE statement

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SELECT COUNT(*) AS BeforeTruncateCount  
FROM Production.TransactionHistory;  
TRUNCATE TABLE Production.TransactionHistory;  
SELECT COUNT(*) AS AfterTruncateCount  
FROM Production.TransactionHistory;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Leave **Microsoft SQL Server Management Studio** open for the next exercise.

Results: After this exercise, you should have created and executed a DELETE statement using the WHERE syntax and created and executed a DELETE statement using the FROM syntax. You should have also created and executed a DELETE statement using the OUTPUT syntax and then created and executed a TRUNCATE TABLE statement.

Exercise 3: Updating Data in Tables

► Task 1: Create an UPDATE statement using the SET syntax

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

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

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 2: Create an UPDATE statement using the WHERE syntax

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

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

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 3: Create an UPDATE statement using the FROM syntax

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

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

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 4: Create an UPDATE statement using the OUTPUT syntax

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
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.BusinessEntityID,  
    DELETED.VacationHours,  
    INSERTED.VacationHours,  
    INSERTED.ModifiedDate  
INTO @MyTableVar;
```

```
SELECT EmpID, OldVacationHours, NewVacationHours, ModifiedDate  
FROM @MyTableVar;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Leave **Microsoft SQL Server Management Studio** open for the next exercise.

Results: After this exercise you should have created and executed an UPDATE statement using the SET syntax, created and executed an UPDATE statement using the WHERE IN (SELECT) syntax, and created and executed an UPDATE statement using the FROM syntax. You should have also created and executed an UPDATE statement using the OUTPUT syntax.

Exercise 4: Working with Transactions

► Task 1: Create a simple transaction

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
BEGIN TRANSACTION CandidateDelete;
DELETE FROM HumanResources.JobCandidate
    WHERE JobCandidateID = 13;
COMMIT TRANSACTION CandidateDelete;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 2: Set a transaction isolation level

1. On the toolbar, click **New Query**.
2. In the new, blank query window type the following T-SQL statement:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;
SELECT * FROM HumanResources.EmployeePayHistory;
SELECT * FROM HumanResources.Department;
COMMIT TRANSACTION;
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Turn off the 2778A-NY-SQL-01 virtual machine and discard any changes.

Results: After this exercise, you should have created a simple transaction. You should have also set the transaction isolation level for a new transaction.

Module 7 Querying Metadata, XML, and Full-Text Indexes

Lab: Querying Metadata, XML, and Full-Text Indexes

Exercise 1: Querying Metadata

► Task 1: Launch SQL Server Management Studio

1. In the Lab Launcher, next to **2778A-NY-SQL-01**, click **Launch**.
2. Log on as **Administrator** with the password of **Pa\$\$w0rd**.
3. Click **Start | All Programs | Microsoft SQL Server 2008**, and then click **SQL Server Management Studio**.
4. In the **Connect to Server** dialog box, verify that **Server type** is set to **Database Engine** and **Server name** is set to **NY-SQL-01**, and then click **Connect**.
5. Maximize **Microsoft SQL Server Management Studio**.
6. In Object Explorer, ensure that **NY-SQL-01 (SQL Server 10.0.1600 - NY-SQL-01\Administrator)** is expanded, and then expand **Databases**.

► Task 2: Query metadata using system catalog views

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
SELECT c.name
FROM sys.columns c
JOIN sys.tables t ON t.object_id = c.object_id
WHERE t.name = 'Person'
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 3: Query metadata using the Information Schema

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
SELECT *
FROM INFORMATION_SCHEMA.columns
WHERE TABLE_NAME = 'Address'
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 4: Query metadata using Dynamic Management Views

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
SELECT count(*), Command, AVG(total_elapsed_time) AS AvgTime
FROM sys.dm_exec_requests
GROUP BY Command
```

3. On the toolbar, click **Execute**.

4. Review the results of the query and then close the query editor without saving the query.

► **Task 5: Query metadata using System Stored Procedures**

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
EXEC sp_columns @table_name = N'Employee',  
    @table_owner = N'HumanResources';
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

Results: After this exercise, you should have launched SQL Server Management Studio and queried metadata using system catalog views, the information schema, dynamic management views, and system stored procedures.

Exercise 2: Querying XML Data

► Task 1: Creating XML output using FOR XML

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
SELECT CustomerID, SalesOrderID, Status  
FROM Sales.SalesOrderHeader OrderHeader  
ORDER BY CustomerID  
FOR XML AUTO
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 2: Querying XML data using OPENXML

1. In the SQL Server Management Studio **File** menu, click **Open | File**. The **File Open** dialog appears.
2. Select **E:\Mod07\Labfiles\Exercise2Task2**, and then click **Open**.
3. Before the SELECT statement, type the following statement:

```
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc;
```

4. On the toolbar, click **Execute**.
5. Review the results of the query and then close the query editor without saving the query.

Results: After this exercise, you should have created a query of relational data and produced the output in XML format. You should have also created a query that converts data in XML format to a relational format.

Exercise 3: Creating and Querying Full-Text Indexes

► Task 1: Creating a full-text index

1. In the Object Explorer, expand **NY-SQL-01 | Databases | AdventureWorks2008 | Storage**.
2. In the Object Explorer, right-click **Full Text Catalogs** and click **New Full-Text Catalog**. The **New Full-Text Catalog** dialog appears.
3. In the **Full-text catalog name** field, type **ProductDescriptionCatalog**.
4. In the **Owner** field, type **dbo**.
5. Click **OK**.
6. In the Object Explorer, expand **Full Text Catalogs**.
7. Right-click **ProductDescriptionCatalog**, and then click **Properties**.
8. In **Select a page**, click **Tables/Views**.
9. In the **All eligible table/view objects in this database** list, select **Production.ProductDescription**.
10. Click **->** to add the table to the Table/view objects assigned to the catalog list.
11. In the **Eligible** columns list, select **Description**.
12. In **Select a page**, click **Population Schedule**.
13. Click **New**.
14. In the **Name** field, type **EveryHour**.
15. In the **Schedule** type list, select **Recurring**.
16. In the **Occurs** list, ensure that **Daily** is selected.
17. In the **Daily frequency** list, select **Occurs every 1 hour**.
18. In the **Starting at** field, enter **12:01:00 AM**.
19. In the **Ending at** field, enter **11:59:00 PM**.
20. Click **OK**.
21. In the **Full-Text Catalog Properties** dialog, click **OK**.
22. In the Object Explorer, right-click on **ProductDescriptionCatalog**, and then click **Rebuild**.
23. In the **Rebuild Full-Text Catalog** dialog, click **OK**.
24. Click **Close**.

► Task 2: Querying the full-text index using FREETEXT

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
SELECT *
FROM Production.ProductDescription
WHERE FREETEXT>Description, 'lightest')
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 3: Querying the full-text index using CONTAINS

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
SELECT *
FROM Production.ProductDescription
WHERE CONTAINS>Description, 'lightest NEAR best')
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Turn off 2778A-NY-SQL-01 virtual machine and discard any changes.

Results: After this exercise, you should have created a new full-text Index and queried that Index using both the FREETEXT and CONTAINS predicates.

Module 8: Using Programming Objects for Data Retrieval

Lab: Using Programming Objects for Data Retrieval

Exercise 1: Creating Views

► Task 1: Launch SQL Server Management Studio

1. In the Lab Launcher, next to **2778A-NY-SQL-01**, click **Launch**.
2. Log on as **Administrator** with the password of **Pa\$\$w0rd**.
3. Click **Start | All Programs | Microsoft SQL Server 2008**, and then click **SQL Server Management Studio**.
4. In the **Connect to Server** dialog box, verify that **Server type** is set to **Database Engine** and **Server name** is set to **NY-SQL-01**, and then click **Connect**.
5. Maximize Microsoft SQL Server Management Studio.
6. In Object Explorer, ensure that **NY-SQL-01 (SQL Server 10.0.1600 - NY-SQL-01\Administrator)** is expanded and then expand **Databases**.

► Task 2: Create a view to select only the ID and Name columns from the Person table

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
CREATE VIEW [dbo].[vwPersonName] AS  
  
    SELECT BusinessEntityID,  
          FirstName,  
          MiddleName,  
          LastName  
    FROM Person.Person  
  
GO
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Right-click the **AdventureWorks2008** database, and then click **New Query**.
6. In the new, blank query window, type the following T-SQL statement:

```
SELECT * FROM vwPersonName
```

7. On the toolbar, click **Execute**.
8. Review the results of the query and then close the query editor without saving the query.

Results: After this exercise, you should have created a view to restrict the number of columns returned for a table.

Exercise 2: Creating User-Defined Functions

► Task 1: Create a scalar valued user-defined function

1. Right-click the **AdventureWorks2008** database and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
CREATE FUNCTION fnGetDate
(
    @input datetime
)
RETURNS varchar(10)
AS
BEGIN
    DECLARE @result varchar(10)

    SET @result = CONVERT(varchar(10), @input, 103)

    RETURN @result
END
GO
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Right-click the **AdventureWorks2008** database, and then click **New Query**.
6. In the new, blank query window, type the following T-SQL statement:

```
SELECT GETDATE()
SELECT dbo.fnGetDate(GETDATE())
```

7. On the toolbar, click **Execute**.
8. Review the results of the query and then close the query editor without saving the query.

► Task 2: Create an in-line table-valued user-defined function

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
CREATE FUNCTION fnGetVendorPurchaseOrders
(
    @VendorID int
)
RETURNS TABLE
AS
RETURN
(
    SELECT PurchaseOrderID,
           RevisionNumber,
           [Status],
           OrderDate,
           ShipDate,
           SubTotal,
           TotalDue
    FROM Purchasing.PurchaseOrderHeader
    WHERE VendorID = @VendorID
)
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Right-click the **AdventureWorks2008** database, and then click **New Query**.
6. In the new, blank query window, type the following T-SQL statement:

```
SELECT * FROM fnGetVendorPurchaseOrders(1624)
```

7. On the toolbar, click **Execute**.
8. Review the results of the query and then close the query editor without saving the query.

Results: After this exercise, you should have launched SQL Server Management Studio and created both a scalar and table-valued user-defined function.

Exercise 3: Creating Stored Procedures

- **Task 1: Create a stored procedure to select the Name columns from the Person table for a given ID**

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
CREATE Procedure pGetPersonName
    @PersonID nvarchar(15)
AS
    SELECT BusinessEntityID,
        FirstName,
        MiddleName,
        LastName
    FROM dbo.vwPersonName
    WHERE BusinessEntityID = @PersonID
GO
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

Results: After this exercise, you should have created a stored procedure that returns a person's name using their ID.

Exercise 4: Writing Distributed Queries

► Task 1: Enabling distributed queries

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
sp_configure 'show advanced options', 1  
reconfigure  
  
GO  
  
sp_configure 'Ad Hoc Distributed Queries', 1  
reconfigure
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 2: Executing an ad hoc distributed query against a Microsoft Office Excel spreadsheet

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
SELECT * FROM OPENROWSET('Microsoft.Jet.OLEDB.4.0',  
'Excel 8.0;Database=E:\Mod08\Labfiles\SalesSummary2008.xls',  
'SELECT Country, TotalSales FROM [Sheet1$]')
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 3: Create a linked server to a Microsoft Office Excel spreadsheet

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
EXEC sp_addlinkedserver 'SalesSummary',  
    'Jet 4.0',  
    'Microsoft.Jet.OLEDB.4.0',  
    'E:\Mod08\Labfiles\SalesSummary2008.xls',  
    NULL,  
    'Excel 8.0'
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.

► Task 4: Query the linked Microsoft Office Excel spreadsheet

1. Right-click the **AdventureWorks2008** database, and then click **New Query**.
2. In the new, blank query window, type the following T-SQL statement:

```
SELECT * FROM SalesSummary...Sheet1$
```

3. On the toolbar, click **Execute**.
4. Review the results of the query and then close the query editor without saving the query.
5. Turn off the 2778A-NY-SQL-01 virtual machine and discard any changes.

Results: After this exercise, you should have used an ad hoc distributed query to query data from a spreadsheet. You have also created a linked server and queried it using a four part name.

Module 9: Using Advanced Techniques

Lab: Using Advanced Techniques

Exercise 1: Using Execution Plans

- ▶ **Task 1: Start the 2778A-NY-SQL-01 virtual machine and log on as Student**
 1. In the Lab Launcher, next to **2778A-NY-SQL-01**, click **Launch**.
 2. Log on to **NY-SQL-01** as **Student** using the password **Pa\$\$w0rd**.
- ▶ **Task 2: View an estimated execution plan**
 1. On the desktop, click **Start**, point to **All Programs**, point to **Microsoft SQL Server 2008**, and then click **SQL Server Management Studio**.
 2. The **Connect to Server** dialog box appears. Click **Connect**.
 3. On the **File** menu, point to **Open** and then click **File**.
 4. The **Open File** dialog box appears. Browse to **E:\MOD09\Labfiles\Starter\Exercise01**.
 5. Click **Order-ProductID.sql** and then click **Open**.
 6. In the database list on the toolbar, click **AdventureWorks2008**.
 7. In the query pane, right-click anywhere on the query and then click **Display Estimated Execution Plan**.
 8. On the **Execution Plan** tab, take note of the flow of the execution plan and the cost of the operations.
- ▶ **Task 3: View an actual execution plan**
 1. In the query pane, right-click anywhere on the query, and then click **Include Actual Execution Plan**.
 2. On the toolbar, click **Execute**.
 3. In the **Results** pane, click the **Execution Plan** tab.
 4. Take note of the flow of the execution plan and the cost of the operations.
- ▶ **Task 4: Save an execution plan**
 1. Right-click anywhere in the **Execution Plan** tab, and then click **Save Execution Plan As**.
 2. The **Save As** dialog box appears. Browse to **E:\MOD09\Labfiles\Starter**.
 3. In the **File name** field, type **Order-ProductID** and then click **Save**.
 4. Right-click anywhere in the **Execution Plan** tab, and then click **Show Execution Plan XML**.
 5. On the **File** menu, click **Save Execution Plan.xml As**.
 6. The **Save File As** dialog box appears. Browse to **E:\MOD09\Labfiles\Starter**.
 7. In the **File name** field, type **Order-ProductID.xml**, and then click **Save**.
 8. Close the **Order-ProductID.xml** pane.

Results: After this exercise, you should have viewed estimated and actual execution plans for a query, and saved an execution plan to an XML file.

Exercise 2: Converting Data Types

► Task 1: Use implicit conversions

1. In Object Explorer, expand Database | AdventureWorks2008 | Tables | HumanResources.Employee | Columns.
2. Notice that the SalariedFlag column has a data type of bit.
3. Right-click AdventureWorks2008, and then click New Query.
4. In the query pane, type the following query:

```
SELECT NationalIDNumber, LoginID, SalariedFlag  
FROM  
HumanResources.Employee  
WHERE  
SalariedFlag = 'False'
```

5. On the toolbar, click **Execute**.
6. In the **Results** pane, review the result set and noticed that the values in the **SalariedFlag** column are **0**.
7. In **Object Explorer**, expand **AdventureWorks2008 | Tables | Production.Product | Columns**.
8. Notice that the **StandardCost** and **ListPrice** columns have a data type of **money**.
9. Right-click **AdventureWorks2008**, and then click **New Query**.
10. In the query pane, type the following query:

```
DECLARE @NewListPrice nvarchar(10) = '39.99'  
DECLARE @NewStandardCost float = 12.45  
  
UPDATE Production.Product  
SET ListPrice = @NewListPrice, StandardCost = @NewStandardCost  
WHERE ProductID = '2'
```

11. On the toolbar, click **Execute**.
12. On the toolbar, click **New Query**.
13. In the query pane, type the following query:

```
SELECT Name, StandardCost, ListPrice  
FROM  
Production.Product  
WHERE  
ProductID = '2'
```

14. On the toolbar click, **Execute**.
15. In the results pane, notice that the **StandardCost** and **ListPrice** columns have been updated.
16. On the **File** menu, point to **Open**, and then click **File**.
17. The **Open File** dialog box appears. Browse to **E:\MOD09\Labfiles\Starter\Exercise02**.
18. Click **UpdateDemographics.sql**, and then click **Open**.
19. In the query pane, notice that the **@NewDemographics** variable has a data type of **nvarchar**, and contains a valid XML string.
20. On the toolbar, click **Execute**.

21. On the toolbar, click **New Query**.
22. In the query pane, type the following query:

```
SELECT FirstName, MiddleName, LastName, Demographics
FROM
Person.Person
WHERE
BusinessEntityID = 15
```

23. On the toolbar, click **Execute**.
24. In the **Results** pane, click the link in the **Demographics** column.
25. The **Demographics.xml** window opens. Notice that the string has been converted and stored as XML.
26. Close the **Demographics.xml** window.

► Task 2: Use explicit conversions

1. On the **File** menu, point to **Open** and then click **File**.
2. The **Open File** dialog box appears. Browse to **E:\MOD09\Labfiles\Starter\Exercise02**.
3. Click **EmployeePayRate.sql**, and then click **Open**.
4. On the toolbar, click **Execute**.
5. In the **Messages** pane, take note of the error that occurs.
6. In the query pane, modify the line beginning with **SELECT** so that it reads as follows:

```
SELECT (p.FirstName + ' ' + p.LastName + ' ' + CONVERT(nvarchar, s.Rate))
```

7. On the toolbar, click **Execute**.
8. In the **Results** pane, notice that there is a single column of employees and pay rates combined.
9. On the **File** menu, point to **Open**, and then click **File**.
10. The **Open File** dialog box appears. Browse to **E:\MOD09\Labfiles\Starter\Exercise02**.
11. Click **PayRateLastChange.sql**, and then click **Open**.
12. On the toolbar, click **Execute**.
13. In the **Results** pane, notice that the query runs successfully, but the last column does not show both the pay rate and date.
14. In the query pane, modify the line beginning with **SELECT** so that it reads as follows:

```
SELECT p.FirstName, p.LastName, (CAST(s.Rate AS nvarchar) + ' ' + s.RateChangeDate)
```

15. On the toolbar, click **Execute**.
16. In the **Messages** pane, take note of the error.
17. In the query pane, modify the line beginning with **SELECT** so that it reads as follows:

```
SELECT p.FirstName, p.LastName, (CAST(s.Rate AS nvarchar) + ' ' +
CAST(s.RateChangeDate as nvarchar))
```

18. On the toolbar, click **Execute**.
19. In the **Results** pane, notice that the query now succeeds and includes the pay rate and date in the last column.

20. Close the **PayRateChange.sql** file. Click **No** when prompted to save changes.

Results: After this exercise, you should have converted several different data types to other data types using explicit and implicit conversion.

Exercise 3: Implementing a Hierarchy

► Task 1: Create a hierarchy

1. On the **File** menu, point to **Open** and then click **Project/Solution**.
2. The **Open Project** dialog box appears. Browse to **E:\MOD09\Labfiles\Starter\Exercise03\SalesTerritory**.
3. Click **SalesTerritory.ssmssln**, and then click **Open**.
4. In **Solution Explorer**, expand **Queries**, and then double-click **CreateTerritoryTable.sql**.
5. On the toolbar, click **Execute**.
6. In Solution Explorer, double-click **CreateTableIndex.sql**.
7. On the toolbar, click **Execute**.
8. In Solution Explorer, double-click **InsertRootTerritory.sql**.
9. On the toolbar, click **Execute**.
10. In Solution Explorer, double-click **InsertChildTerritory.sql**.
11. On the toolbar, click **Execute**.
12. In Solution Explorer, double-click **CreateTerritorySP.sql**.
13. On the toolbar, click **Execute**.
14. In Solution Explorer, double-click **PopulateTerritoryTable.sql**.
15. On the toolbar, click **Execute**.

► Task 2: Query a hierarchy

1. In Solution Explorer, double-click **GetTerritoryOrg.sql**.
2. On the toolbar, click **Execute**.
3. In the **Results** table, notice the results and how the rows relate to one another.
4. In Solution Explorer, double-click **GetNATerritories.sql**.
5. On the toolbar, click **Execute**.
6. In the **Results** pane, notice that only the territories that report to North America are shown.

Results: After this exercise, you should have created a hierarchy to store hierarchical data using the **hierarchyid** data type.

Exercise 4: Using Cursors and Set-Based Queries

► Task 1: Update the ListPrice column

1. On the toolbar, click **New Query**.
2. In the query pane, type the following query:

```
SELECT ProductID, Name, StandardCost, ListPrice FROM Production.Product
```

3. On the toolbar, click **Execute**.
4. In the **Results** pane, review the **ListPrice** and **StandardCost** columns. Keep the query pane open.
5. On the **File** menu, point to **Open**, and then click **File**.
6. The **Open File** dialog box appears. Browse to **E:\MOD09\Labfiles\Starter\Exercise04**.
7. Click **UpdateListPriceCursor.sql**, and then click **Open**.
8. On the toolbar, click **Execute**.
9. Switch back to the SELECT query above and then on the toolbar, click **Execute**.
10. In the **Results** pane, verify that the **ListPrice** column has been updated. Keep the query pane open.

► Task 2: Update the StandardCost column

1. On the toolbar, click **New Query**.
2. In the query pane, type the following query:

```
UPDATE Production.Product  
SET StandardCost = 7.99
```

3. On the toolbar click **Execute**.
4. Switch back to the SELECT query above, and then on the toolbar, click **Execute**.
5. In the **Results** pane, verify that the **StandardCost** column has been updated.
6. Close SQL Server Management Studio. Click **No** if prompted to save changes.
7. Turn off the virtual machine and discard changes.

Results: After this exercise, you should have written a cursor to perform operations on a result set, and rewritten a cursor to use a set-based approach.

Notes

Notes