Microsoft

OFFICIAL MICROSOFT LEARNING PRODUCT

20461D

Querying Microsoft® SQL Server® 2014

MCT USE ONLY. STUDENT USE PROHIBITED

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.

© 2016 Microsoft Corporation. All rights reserved.

Microsoft and the trademarks listed at http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners

Product Number: 20461D

Part Number (if applicable): X20-90569

Released: 01/2016

MCT USE ONLY. STUDENT USE PROHIBITED

MICROSOFT LICENSE TERMS
MICROSOFT INSTRUCTOR-LED COURSEWARE

These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to your use of the content accompanying this agreement which includes the media on which you received it, if any. These license terms also apply to Trainer Content and any updates and supplements for the Licensed Content unless other terms accompany those items. If so, those terms apply.

BY ACCESSING, DOWNLOADING OR USING THE LICENSED CONTENT, YOU ACCEPT THESE TERMS. IF YOU DO NOT ACCEPT THEM, DO NOT ACCESS, DOWNLOAD OR USE THE LICENSED CONTENT.

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

1. DEFINITIONS.

a. "Authorized Learning Center" means a Microsoft IT Academy Program Member, Microsoft Learning Competency Member, or such other entity as Microsoft may designate from time to time.

b. "Authorized Training Session" means the instructor-led training class using Microsoft Instructor-Led Courseware conducted by a Trainer at or through an Authorized Learning Center.

c. "Classroom Device" means one (1) dedicated, secure computer that an Authorized Learning Center owns or controls that is located at an Authorized Learning Center's training facilities that meets or exceeds the hardware level specified for the particular Microsoft Instructor-Led Courseware.

d. "End User" means an individual who is (i) duly enrolled in and attending an Authorized Training Session or Private Training Session, (ii) an employee of a MPN Member, or (iii) a Microsoft full-time employee.

e. "Licensed Content" means the content accompanying this agreement which may include the Microsoft Instructor-Led Courseware or Trainer Content.

f. "Microsoft Certified Trainer" or "MCT" means an individual who is (i) engaged to teach a training session to End Users on behalf of an Authorized Learning Center or MPN Member, and (ii) currently certified as a Microsoft Certified Trainer under the Microsoft Certification Program.

g. "Microsoft Instructor-Led Courseware" means the Microsoft-branded instructor-led training course that educates IT professionals and developers on Microsoft technologies. A Microsoft Instructor-Led Courseware title may be branded as MOC, Microsoft Dynamics or Microsoft Business Group courseware.

h. "Microsoft IT Academy Program Member" means an active member of the Microsoft IT Academy Program.

i. "Microsoft Learning Competency Member" means an active member of the Microsoft Partner Network program in good standing that currently holds the Learning Competency status.

j. "MOC" means the "Official Microsoft Learning Product" instructor-led courseware known as Microsoft Official Course that educates IT professionals and developers on Microsoft technologies.

k. "MPN Member" means an active Microsoft Partner Network program member in good standing.

MCT USE ONLY. STUDENT USE PROHIBITED

l. "Personal Device" means one (1) personal computer, device, workstation or other digital electronic device that you personally own or control that meets or exceeds the hardware level specified for the particular Microsoft Instructor-Led Courseware.

m. "Private Training Session" means the instructor-led training classes provided by MPN Members for corporate customers to teach a predefined learning objective using Microsoft Instructor-Led Courseware. These classes are not advertised or promoted to the general public and class attendance is restricted to individuals employed by or contracted by the corporate customer.

n. "Trainer" means (i) an academically accredited educator engaged by a Microsoft IT Academy Program Member to teach an Authorized Training Session, and/or (ii) a MCT.

o. "Trainer Content" means the trainer version of the Microsoft Instructor-Led Courseware and additional supplemental content designated solely for Trainers' use to teach a training session using the Microsoft Instructor-Led Courseware. Trainer Content may include Microsoft PowerPoint presentations, trainer preparation guide, train the trainer materials, Microsoft One Note packs, classroom setup guide and Pre-release course feedback form. To clarify, Trainer Content does not include any software, virtual hard disks or virtual machines.

2. USE RIGHTS. The Licensed Content is licensed not sold. The Licensed Content is licensed on a ***one copy per user basis***, such that you must acquire a license for each individual that accesses or uses the Licensed Content.

2.1 Below are five separate sets of use rights. Only one set of rights apply to you.

a. **If you are a Microsoft IT Academy Program Member:**

i. Each license acquired on behalf of yourself may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.

ii. For each license you acquire on behalf of an End User or Trainer, you may either:

1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User who is enrolled in the Authorized Training Session, and only immediately prior to the commencement of the Authorized Training Session that is the subject matter of the Microsoft Instructor-Led Courseware being provided, **or**
2. provide one (1) End User with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
3. provide one (1) Trainer with the unique redemption code and instructions on how they can access one (1) Trainer Content,

provided you comply with the following:

iii. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,

iv. you will ensure each End User attending an Authorized Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Authorized Training Session,

v. you will ensure that each End User provided with the hard-copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,

vi. you will ensure that each Trainer teaching an Authorized Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Authorized Training Session,

MCT USE ONLY. STUDENT USE PROHIBITED

vii. you will only use qualified Trainers who have in-depth knowledge of and experience with the Microsoft technology that is the subject of the Microsoft Instructor-Led Courseware being taught for all your Authorized Training Sessions,
viii. you will only deliver a maximum of 15 hours of training per week for each Authorized Training Session that uses a MOC title, and
ix. you acknowledge that Trainers that are not MCTs will not have access to all of the trainer resources for the Microsoft Instructor-Led Courseware.

b. **If you are a Microsoft Learning Competency Member**:
   i. Each license acquired on behalf of yourself may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
   ii. For each license you acquire on behalf of an End User or Trainer, you may either:
      1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User attending the Authorized Training Session and only immediately prior to the commencement of the Authorized Training Session that is the subject matter of the Microsoft Instructor-Led Courseware provided, **or**
      2. provide one (1) End User attending the Authorized Training Session with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
      3. you will provide one (1) Trainer with the unique redemption code and instructions on how they can access one (1) Trainer Content,

   provided you comply with the following:
   iii. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
   iv. you will ensure that each End User attending an Authorized Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Authorized Training Session,
   v. you will ensure that each End User provided with a hard-copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,
   vi. you will ensure that each Trainer teaching an Authorized Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Authorized Training Session,
   vii. you will only use qualified Trainers who hold the applicable Microsoft Certification credential that is the subject of the Microsoft Instructor-Led Courseware being taught for your Authorized Training Sessions,
   viii. you will only use qualified MCTs who also hold the applicable Microsoft Certification credential that is the subject of the MOC title being taught for all your Authorized Training Sessions using MOC,
   ix. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and
   x. you will only provide access to the Trainer Content to Trainers.

MCT USE ONLY. STUDENT USE PROHIBITED

c. **If you are a MPN Member**:

i. Each license acquired on behalf of yourself may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.

ii. For each license you acquire on behalf of an End User or Trainer, you may either:

1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User attending the Private Training Session, and only immediately prior to the commencement of the Private Training Session that is the subject matter of the Microsoft Instructor-Led Courseware being provided, **or**
2. provide one (1) End User who is attending the Private Training Session with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
3. you will provide one (1) Trainer who is teaching the Private Training Session with the unique redemption code and instructions on how they can access one (1) Trainer Content,

provided you comply with the following:

iii. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,

iv. you will ensure that each End User attending an Private Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Private Training Session,

v. you will ensure that each End User provided with a hard copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,

vi. you will ensure that each Trainer teaching an Private Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Private Training Session,

vii. you will only use qualified Trainers who hold the applicable Microsoft Certification credential that is the subject of the Microsoft Instructor-Led Courseware being taught for all your Private Training Sessions,

viii. you will only use qualified MCTs who hold the applicable Microsoft Certification credential that is the subject of the MOC title being taught for all your Private Training Sessions using MOC,

ix. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and

x. you will only provide access to the Trainer Content to Trainers.

d. **If you are an End User:**

For each license you acquire, you may use the Microsoft Instructor-Led Courseware solely for your personal training use. If the Microsoft Instructor-Led Courseware is in digital format, you may access the Microsoft Instructor-Led Courseware online using the unique redemption code provided to you by the training provider and install and use one (1) copy of the Microsoft Instructor-Led Courseware on up to three (3) Personal Devices. You may also print one (1) copy of the Microsoft Instructor-Led Courseware. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.

e. **If you are a Trainer.**

i. For each license you acquire, you may install and use one (1) copy of the Trainer Content in the form provided to you on one (1) Personal Device solely to prepare and deliver an Authorized Training Session or Private Training Session, and install one (1) additional copy on another Personal Device as a backup copy, which may be used only to reinstall the Trainer Content. You may not install or use a copy of the Trainer Content on a device you do not own or control. You may also print one (1) copy of the Trainer Content solely to prepare for and deliver an Authorized Training Session or Private Training Session.

MCT USE ONLY. STUDENT USE PROHIBITED

ii. You may customize the written portions of the Trainer Content that are logically associated with instruction of a training session in accordance with the most recent version of the MCT agreement. If you elect to exercise the foregoing rights, you agree to comply with the following: (i) customizations may only be used for teaching Authorized Training Sessions and Private Training Sessions, and (ii) all customizations will comply with this agreement. For clarity, any use of "*customize"* refers only to changing the order of slides and content, and/or not using all the slides or content, it does not mean changing or modifying any slide or content.

2.2 **Separation of Components.** The Licensed Content is licensed as a single unit and you may not separate their components and install them on different devices.

2.3 **Redistribution of Licensed Content**. Except as expressly provided in the use rights above, you may not distribute any Licensed Content or any portion thereof (including any permitted modifications) to any third parties without the express written permission of Microsoft.

2.4 **Third Party Notices**. The Licensed Content may include third party code tent that Microsoft, not the third party, licenses to you under this agreement. Notices, if any, for the third party code ntent are included for your information only.

2.5 **Additional Terms**. Some Licensed Content may contain components with additional terms, conditions, and licenses regarding its use. Any non-conflicting terms in those conditions and licenses also apply to your use of that respective component and supplements the terms described in this agreement.

3. LICENSED CONTENT BASED ON PRE-RELEASE TECHNOLOGY. If the Licensed Content's subject matter is based on a pre-release version of Microsoft technology ("**Pre-release**"), then in addition to the other provisions in this agreement, these terms also apply:

a. **Pre-Release Licensed Content.** This Licensed Content subject matter is on the Pre-release version of the Microsoft technology. The technology may not work the way a final version of the technology will and we may change the technology for the final version. We also may not release a final version. Licensed Content based on the final version of the technology may not contain the same information as the Licensed Content based on the Pre-release version. Microsoft is under no obligation to provide you with any further content, including any Licensed Content based on the final version of the technology.

b. **Feedback.** If you agree to give feedback about the Licensed Content to Microsoft, either directly or through its third party designee, 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 technology, Microsoft product, or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its technology, technologies, or products to third parties because we include your feedback in them. These rights survive this agreement.

c. **Pre-release Term**. If you are an Microsoft IT Academy Program Member, Microsoft Learning Competency Member, MPN Member or Trainer, you will cease using all copies of the Licensed Content on the Pre-release technology upon (i) the date which Microsoft informs you is the end date for using the Licensed Content on the Pre-release technology, or (ii) sixty (60) days after the commercial release of the technology that is the subject of the Licensed Content, whichever is earliest ("**Pre-release term**"). Upon expiration or termination of the Pre-release term, you will irretrievably delete and destroy all copies of the Licensed Content in your possession or under your control.

MCT USE ONLY STUDENT USE PROHIBITED

4. **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 allows you to use it in certain ways. Except as expressly permitted in this agreement, you may not:
   - access or allow any individual to access the Licensed Content if they have not acquired a valid license for the Licensed Content,
   - alter, remove or obscure any copyright or other protective notices (including watermarks), branding or identifications contained in the Licensed Content,
   - modify or create a derivative work of any Licensed Content,
   - publicly display, or make the Licensed Content available for others to access or use,
   - copy, print, install, sell, publish, transmit, lend, adapt, reuse, link to or post, make available or distribute the Licensed Content to any third party,
   - work around any technical limitations in the Licensed Content, or
   - reverse engineer, decompile, remove or otherwise thwart any protections or disassemble the Licensed Content except and only to the extent that applicable law expressly permits, despite this limitation.

5. **RESERVATION OF RIGHTS AND OWNERSHIP**. Microsoft reserves all rights not expressly granted to you in this agreement. The Licensed Content is protected by copyright and other intellectual property laws and treaties. Microsoft or its suppliers own the title, copyright, and other intellectual property rights in the Licensed Content.

6. **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.

7. **SUPPORT SERVICES**. Because the Licensed Content is "as is", we may not provide support services for it.

8. **TERMINATION.** Without prejudice to any other rights, Microsoft may terminate this agreement if you fail to comply with the terms and conditions of this agreement. Upon termination of this agreement for any reason, you will immediately stop all use of and delete and destroy all copies of the Licensed Content in your possession or under your control.

9. **LINKS TO THIRD PARTY SITES**. You may link to third party sites through the use of the Licensed Content. The third party sites are not under the control of Microsoft, and Microsoft is not responsible for the contents of any third party sites, any links contained in third party sites, or any changes or updates to third party sites. Microsoft is not responsible for webcasting or any other form of transmission received from any third party sites. Microsoft is providing these links to third party sites to you only as a convenience, and the inclusion of any link does not imply an endorsement by Microsoft of the third party site.

10. **ENTIRE AGREEMENT.** This agreement, and any additional terms for the Trainer Content, updates and supplements are the entire agreement for the Licensed Content, updates and supplements.

11. **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.

MCT USE ONLY. STUDENT USE PROHIBITED

b. Outside the United States. If you acquired the Licensed Content in any other country, the laws of that country apply.

12. 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.

13. DISCLAIMER OF WARRANTY. THE LICENSED CONTENT IS LICENSED "AS-IS" AND "AS AVAILABLE." YOU BEAR THE RISK OF USING IT. MICROSOFT AND ITS RESPECTIVE AFFILIATES 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 AND ITS RESPECTIVE AFFILIATES EXCLUDES ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.

14. LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. YOU CAN RECOVER FROM MICROSOFT, ITS RESPECTIVE AFFILIATES AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO US$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, 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 dues consommateurs, que ce contrat ne peut modifier. La ou elles sont permises par le droit locale, 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.

MCT USE ONLY STUDENT USE PROHIBITED

Elle s'applique également, même si Microsoft connaissait 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.

Revised July 2013

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY

Welcome!

Thank you for taking our training! We've worked together with our Microsoft Certified Partners for Learning Solutions and our Microsoft IT Academies to bring you a world-class learning experience—whether you're a professional looking to advance your skills or a student preparing for a career in IT.

- **Microsoft Certified Trainers and Instructors**—Your instructor is a technical and instructional expert who meets ongoing certification requirements. And, if instructors are delivering training at one of our Certified Partners for Learning Solutions, they are also evaluated throughout the year by students and by Microsoft.

- **Certification Exam Benefits**—After training, consider taking a Microsoft Certification exam. Microsoft Certifications validate your skills on Microsoft technologies and can help differentiate you when finding a job or boosting your career. In fact, independent research by IDC concluded that 75% of managers believe certifications are important to team performance[1]. Ask your instructor about Microsoft Certification exam promotions and discounts that may be available to you.

- **Customer Satisfaction Guarantee**—Our Certified Partners for Learning Solutions offer a satisfaction guarantee and we hold them accountable for it. At the end of class, please complete an evaluation of today's experience. We value your feedback!

We wish you a great learning experience and ongoing success in your career!

Sincerely,

Microsoft Learning
www.microsoft.com/learning

Microsoft® | Learning

[1] *IDC,* Value of Certification: Team Certification and Organizational Performance, *November 2006*

Acknowledgments

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.

Geoff Allix – Lead Content Developer

Geoff Allix is a Microsoft SQL Server subject matter expert and professional content developer at Content Master—a division of CM Group Ltd. As a Microsoft Certified Trainer, Geoff has delivered training courses on SQL Server since version 6.5. Geoff is a Microsoft Certified IT Professional for SQL Server and has extensive experience in designing and implementing database and BI solutions on SQL Server technologies, and has provided consultancy services to organizations seeking to implement and optimize database solutions.

MCT USE ONLY. STUDENT USE PROHIBITED

Contents

Module 1: Introduction to Microsoft SQL Server 2014

Module Overview **1-1**

Lesson 1: The Basic Architecture of SQL Server **1-2**

Lesson 2: SQL Server Editions and Versions **1-6**

Lesson 3: Getting Started with SQL Server Management Studio **1-8**

Lab: Working with SQL Server 2014 Tools **1-15**

Module Review and Takeaways **1-18**

Module 2: Introduction to T-SQL Querying

Module Overview **2-1**

Lesson 1: Introducing T-SQL **2-2**

Lesson 2: Understanding Sets **2-12**

Lesson 3: Understanding Predicate Logic **2-15**

Lesson 4: Understanding the Logical Order of Operations in SELECT Statements **2-17**

Lab: Introduction to T-SQL Querying **2-22**

Module Review and Takeaways **2-25**

Module 3: Writing SELECT Queries

Module Overview **3-1**

Lesson 1: Writing Simple SELECT Statements **3-2**

Lesson 2: Eliminating Duplicates with DISTINCT **3-6**

Lesson 3: Using Column and Table Aliases **3-11**

Lesson 4: Writing Simple CASE Expressions **3-16**

Lab: Writing Basic SELECT Statements **3-19**

Module Review and Takeaways **3-24**

Module 4: Querying Multiple Tables

Module Overview **4-1**

Lesson 1: Understanding Joins **4-2**

Lesson 2: Querying with Inner Joins **4-7**

Lesson 3: Querying with Outer Joins **4-11**

Lesson 4: Querying with Cross Joins and Self Joins **4-15**

Lab: Querying Multiple Tables **4-19**

Module Review and Takeaways **4-24**

MCT USE ONLY. STUDENT USE PROHIBITED

Module 5: Sorting and Filtering Data

Module Overview **5-1**

Lesson 1: Sorting Data **5-2**

Lesson 2: Filtering Data with Predicates **5-6**

Lesson 3: Filtering Data with TOP and OFFSET-FETCH **5-10**

Lesson 4: Working with Unknown Values **5-16**

Lab: Sorting and Filtering Data **5-19**

Module Review and Takeaways **5-24**

Module 6: Working with SQL Server 2014 Data Types

Module Overview **6-1**

Lesson 1: Introducing SQL Server 2014 Data Types **6-2**

Lesson 2: Working with Character Data **6-10**

Lesson 3: Working with Date and Time Data **6-17**

Lab: Working with SQL Server 2014 Data Types **6-23**

Module Review and Takeaways **6-29**

Module 7: Using DML to Modify Data

Module Overview **7-1**

Lesson 1: Adding Data to Tables **7-2**

Lesson 2: Modifying and Removing Data **7-6**

Lesson 3: Generating Numbers **7-11**

Lab: Using DML to Modify Data **7-14**

Module Review and Takeaways **7-16**

Module 8: Using Built-In Functions

Module Overview **8-1**

Lesson 1: Writing Queries with Built-In Functions **8-2**

Lesson 2: Using Conversion Functions **8-8**

Lesson 3: Using Logical Functions **8-13**

Lesson 4: Using Functions to Work with NULL **8-17**

Lab: Using Built-In Functions **8-21**

Module Review and Takeaways **8-25**

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 9: Grouping and Aggregating Data

Module Overview **9-1**

Lesson 1: Using Aggregate Functions **9-2**

Lesson 2: Using the GROUP BY Clause **9-9**

Lesson 3: Filtering Groups with HAVING **9-14**

Lab: Grouping and Aggregating Data **9-17**

Module Review and Takeaways **9-23**

Module 10: Using Subqueries

Module Overview **10-1**

Lesson 1: Writing Self-Contained Subqueries **10-2**

Lesson 2: Writing Correlated Subqueries **10-7**

Lesson 3: Using the EXISTS Predicate with Subqueries **10-11**

Lab: Using Subqueries **10-14**

Module Review and Takeaways **10-19**

Module 11: Using Table Expressions

Module Overview **11-1**

Lesson 1: Using Views **11-2**

Lesson 2: Using Inline TVFs **11-5**

Lesson 3: Using Derived Tables **11-9**

Lesson 4: Using CTEs **11-15**

Lab: Using Table Expressions **11-18**

Module Review and Takeaways **11-25**

Module 12: Using Set Operators

Module Overview **12-1**

Lesson 1: Writing Queries with the UNION Operator **12-2**

Lesson 2: Using EXCEPT and INTERSECT **12-6**

Lesson 3: Using APPLY **12-9**

Lab: Using Set Operators **12-13**

Module Review and Takeaways **12-18**

Module 13: Using Window Ranking, Offset, and Aggregate Functions

Module Overview **13-1**

Lesson 1: Creating Windows with OVER **13-2**

Lesson 2: Exploring Window Functions **13-8**

Lab: Using Window Ranking, Offset, and Aggregate Functions **13-16**

Module Review and Takeaways **13-20**

Module 14: Pivoting and Grouping Sets

Module Overview **14-1**

Lesson 1: Writing Queries with PIVOT and UNPIVOT **14-2**

Lesson 2: Working with Grouping Sets **14-7**

Lab: Pivoting and Grouping Sets **14-13**

Module Review and Takeaways **14-19**

Module 15: Executing Stored Procedures

Module Overview **15-1**

Lesson 1: Querying Data with Stored Procedures **15-2**

Lesson 2: Passing Parameters to Stored Procedures **15-5**

Lesson 3: Creating Simple Stored Procedures **15-9**

Lesson 4: Working with Dynamic SQL **15-12**

Lab: Executing Stored Procedures **15-16**

Module Review and Takeaways **15-22**

Module 16: Programming with T-SQL

Module Overview **16-1**

Lesson 1: T-SQL Programming Elements **16-2**

Lesson 2: Controlling Program Flow **16-7**

Lab: Programming with T-SQL **16-11**

Module Review and Takeaways **16-17**

Module 17: Implementing Error Handling

Module Overview **17-1**

Lesson 1: Implementing T-SQL Error Handling **17-2**

Lesson 2: Implementing Structured Exception Handling **17-7**

Lab: Implementing Error Handling **17-11**

Module Review and Takeaways **17-14**

MCT USE ONLY. STUDENT USE PROHIBITED

Module 18: Implementing Transactions

Module Overview 18-1

Lesson 1: Transactions and the Database Engine 18-2

Lesson 2: Controlling Transactions 18-7

Lab: Implementing Transactions 18-12

Module Review and Takeaways 18-16

Appendix A: Improving Query Performance

Module Overview 19-1

Lesson 1: Factors in Query Performance 19-2

Lesson 2: Displaying Query Performance Data 19-10

Lab: Improving Query Performance 19-16

Module Review and Takeaways 19-20

Appendix B: Querying SQL Server Metadata

Module Overview 20-1

Lesson 1: Querying System Catalog Views and Functions 20-2

Lesson 2: Executing System Stored Procedures 20-7

Lesson 3: Querying Dynamic Management Objects 20-11

Lab: Querying SQL Server Metadata 20-15

Module Review and Takeaways 20-20

Lab Answer Keys

Module 1 Lab: Working with SQL Server 2014 Tools L01-1

Module 2 Lab: Introduction to T-SQL Querying L02-1

Module 3 Lab: Writing Basic SELECT Statements L03-1

Module 4 Lab: Querying Multiple Tables L04-1

Module 5 Lab: Sorting and Filtering Data L05-1

Module 6 Lab: Working with SQL Server 2014 Data Types L06-1

Module 7 Lab: Using DML to Modify Data L07-1

Module 8 Lab: Using Built-In Functions L08-1

Module 9 Lab: Grouping and Aggregating Data L09-1

Module 10 Lab: Using Subqueries L10-1

Module 11 Lab: Using Table Expressions L11-1

Module 12 Lab: Using Set Operators L12-1

Module 13 Lab: Using Window Ranking, Offset, and Aggregate Functions L13-1

Module 14 Lab: Pivoting and Grouping Sets L14-1

MCT USE ONLY. STUDENT USE PROHIBITED

Module 15 Lab: Executing Stored Procedures **L15-1**

Module 16 Lab: Programming with T-SQL **L16-1**

Module 17 Lab: Implementing Error Handling **L17-1**

Module 18 Lab: Implementing Transactions **L18-1**

Appendix A Lab: Improving Query Performance **L19-1**

Appendix B Lab: Querying SQL Server Metadata **L20-1**

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

About This Course

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

Course Description

This 5-day instructor led course provides students with the technical skills required to write basic Transact-SQL queries for Microsoft SQL Server 2014. This course is the foundation for all SQL Server-related disciplines; namely, Database Administration, Database Development and Business Intelligence. This course helps people prepare for exam 70-461: Writing Queries Using Microsoft® SQL Server® 2012 Transact-SQL.

Audience

This course is intended for Database Administrators, Database Developers, and Business Intelligence professionals. The course will very likely be well attended by SQL power users who aren't necessarily database-focused or plan on taking the exam; namely, report writers, business analysts and client application developers.

Student Prerequisites

This course requires that you meet the following prerequisites:

Before attending this course, students must have:

- Working knowledge of relational databases.
- Basic knowledge of the Microsoft Windows operating system and its core functionality.

Before attending this course, students should have:

- Basic understanding of virtualization technology (Classroom labs utilize virtual machines)

Course Objectives

After completing this course, students will be able to:

- Describe the basic architecture and concepts of Microsoft SQL Server 2014.
- Understand the similarities and differences between Transact-SQL and other computer languages.
- Write Transact-SQL queries using the SELECT statement.
- Query multiple tables.
- Limit the rows that queries return and control the order in which the rows are displayed.
- Describe SQL Server 2014 data types.
- Update data using Transact-SQL.
- Use Transact-SQL functions.
- Aggregate data using Transact-SQL.
- Nest Transact-SQL queries using subqueries.
- Describe and implement table expressions.
- Compare rows between two input sets.

- Write queries that use window functions.
- Use the PIVOT operator to change the orientation of data.
- Create and execute stored procedures.
- Enhance your T-SQL code with programming elements.
- Implement error handling in Transact-SQL.
- Implement transactions in Transact-SQL.
- Describe techniques to improve query performance.
- Query SQL Server metadata.

Course Outline

This section provides an outline of the course:

Module 1, "Introduction to Microsoft SQL Server 2014"

Module 2, "Introduction to T-SQL Querying"

Module 3, "Writing SELECT Queries"

Module 4, "Querying Multiple Tables"

Module 5, "Sorting and Filtering Data"

Module 6, "Working with SQL Server 2014 Data Types"

Module 7, "Using DML to Modify Data"

Module 8, "Using Built-In Functions"

Module 9, "Grouping and Aggregating Data"

Module 10, "Using Subqueries"

Module 11, "Using Table Expressions"

Module 12, "Using Set Operators"

Module 13, "Using Window Ranking, Offset, and Aggregate Functions"

Module 14, "Pivoting and Grouping Sets"

Module 15, "Executing Stored Procedures"

Module 16, "Programming with T-SQL"

Module 17, "Implementing Error Handling"

Module 18, "Implementing Transactions"

Appendix A, "Improving Query Performance"

Appendix B, "Querying SQL Server Metadata"

MCT USE ONLY. STUDENT USE PROHIBITED

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 fingertips when it's needed.

Course Companion Content on the *http://www.microsoft.com/learning/companionmoc/* Site*:* *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: 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 mcphelp@microsoft.com.

MCT USE ONLY. STUDENT USE PROHIBITED

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 Hyper-V to perform the labs.

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

Virtual machine	Role
20461C-MIA-SQL	Database Server
20461C-MIA-DC	Domain Controller

Software Configuration

The following software is installed on each VM:

- Windows Server® 2012
- Microsoft SQL Server 2014
- Microsoft SharePoint Server 2013
- Microsoft Office 2013
- Microsoft Visual Studio 2012

Course Files

There are files associated with the labs in this course. The lab files are located in the folder D:\Labfiles\LabXX on the 20461C-MIA-SQL virtual machine.

Classroom Setup

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

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.

Course Hardware Level 6+

- Processor: Intel Virtualization Technology (Intel VT) or AMD Virtualization (AMD-V)
- Hard Disk: Dual 120 GB hard disks 7200 RM SATA or better (Striped)
- RAM: 8GB or higher. 16 GB or more is recommended for this course.
- DVD/CD: DVD drive
- Network adapter with Internet connectivity
- Video Adapter/Monitor: 17-inch Super VGA (SVGA)
- Microsoft Mouse or compatible pointing device
- Sound card with amplified speakers

MCT USE ONLY. STUDENT USE PROHIBITED

In addition, the instructor computer must be connected to a projection display device that supports SVGA 1024 x 768 pixels, 16 bit colors.

Note: For the best classroom experience, a computer with solid state disks (SSDs) is recommended. For optimal performance, adapt the instructions below to install the 20461C-MIA-SQL virtual machine on a different physical disk than the other virtual machines to reduce disk contention.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 1

Introduction to Microsoft SQL Server 2014

Contents:

Module Overview	**1-1**
Lesson 1: The Basic Architecture of SQL Server	**1-2**
Lesson 2: SQL Server Editions and Versions	**1-6**
Lesson 3: Getting Started with SQL Server Management Studio	**1-8**
Lab: Working with SQL Server 2014 Tools	**1-15**
Module Review and Takeaways	**1-18**

Module Overview

Before beginning to learn how to write queries with Microsoft SQL Server 2014, it is useful to understand the overall SQL Server database platform, including its basic architecture, the various editions available for SQL Server 2014, and the tools a query writer will use. This module will also prepare you to use SQL Server Management Studio, SQL Server's primary development and administration tool, to connect to SQL Server instances and create, organize, and execute queries.

Objectives

After completing this module, you will be able to:

- Describe the architecture of SQL Server 2014.
- Describe the editions of SQL Server 2014.
- Work with SQL Server Management Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
The Basic Architecture of SQL Server

In this lesson, you will learn about the basic architecture and concepts of Microsoft SQL Server 2014. You will learn how instances, services, and databases interact, and how databases are structured. This will help prepare you to begin working with SQL Server queries in upcoming modules.

Lesson Objectives

After completing this lesson, you will be able to describe:

- Relational databases and, specifically, the role and structure of SQL Server databases.
- The sample database used in this course.
- Client server databases.
- The structure of Transact-SQL queries.

Relational Databases

Typical relational databases comprise of several tables that relate to each other. Each table typically represents a class of entity which might be something tangible, such as an employee, or something intangible, such as a sales order. In this example, a very simple database might have an employee table and an order table, with employees able to place orders.

We find meaningful information by using joins, which are possible when two tables share values. To continue the example, each order has an employee ID for the person who placed the order. It is also possible to join several tables. Let's extend the example and add customers. Now each order also contains a customer ID that can be used to link it to the new customer table. We can now display an employee and their customers by joining all three tables. It is not necessary to display any order information, we can just use this as a bridge between the other two tables.

Databases in SQL Server are containers for data and objects, including tables, views, stored procedures, user accounts, and other management objects. An SQL Server database is always a single logical entity, backed by multiple physical files.

SQL Server supports two types of databases—system and user. TSQL, the sample database you will be using to write queries, is a user database. SQL Server's system databases include:

- master, the system configuration database.
- msdb, the configuration database for the SQL Server Agent service and other system services.
- model, the template for new user databases.
- tempdb, used by the database engine to store temporary data such as work tables. This database is dropped and recreated each time SQL Server restarts. Never store anything you need to depend on in it!

- Resource, a hidden system configuration database that provides system objects to other databases.

Database administrators and developers can create user databases to hold data and objects for applications. You connect to a user database to execute your queries. You will need security credentials to log on to SQL Server and a database account with permissions to access data objects in the database.

About the Course Sample Database

In this course, most of your queries will be written against a sample database named TSQL2014. This is designed as a small, low-complexity database suitable for learning to write T-SQL queries. It contains several types of objects:

- User-defined schemas, which are containers for tables and views. You will learn about schemas later in this course.
- Tables, which relate to other tables via foreign key constraints.
- Views, which display aggregated information.

- TSQL database used in course
- Represents a business with products, customers, orders, suppliers, employees
- Contains sample schemas:
  - HR, Production, Sales
- Contains sample tables, views
- Scripts provided with labs and demos to recreate database if needed

The TSQL2014 database is modeled to resemble a sales-tracking application for a small business. Some of the tables you will use include:

- Sales.Orders, which stores data typically found in the header of an invoice (order ID, customer ID, order date, and so on).
- Sales.OrderDetails, which stores transaction details about each order (parent order ID, product ID, unit price, and so on).
- Sales.Customers, which stores information about customers (company name, contact details, and so on).
- HR.Employees, which stores information about the company's employees (name, birthdate, hire date, and so on).

Other tables are supplied to add context, such as additional product information, to these tables.

Client Server Databases

SQL Server is a client server system. This means that the client software, which includes SQL Server Management Studio and Visual Studio, is separate from the SQL Server database engine.

When client applications send requests to the database engine as T-SQL statements, SQL Server performs all file, memory, and processor utilization on the client's behalf. Clients never directly access database files, unlike in desktop database applications.

- Client software is separate from database engine
- Client software and database engine can be on the same machine or connected over a network
- With correct configuration, databases can access data in other databases over the network

MCT USE ONLY STUDENT USE PROHIBITED

In the course, the client and server are running on the same virtual machine, but in most environments the client software is running on a separate machine to the database engine.

Whether the database engine is local, or you are connecting to it over a network, it makes no difference to the T-SQL code that we write. On the logon screen, you just need to specify the server that you are connecting to.

Because you can connect to instances of SQL Server over a network, you can also refer to other databases in a T-SQL script. To do this, you need to refer to a table, or other object, using its four-part name. This takes the format of **Instance.Database.Schema.Object**. For example, the **orders** table in the **dbo** schema, in the **sales** database on the **MIA-SQL** server's default instance, would be referred to as **MIA-SQL.sales.dbo.orders**. To connect to a remote server in a T-SQL script, you should set up the remote instance as a linked server. In T-SQL you can add a linked server using the **sp_addlinkedserver** stored procedure. Although there are many arguments that can be supplied, in its most straightforward, default, use you could connect to the server in the previous example using the statement **exec sp_addlinkedserver in'MIA-SQL'**.

Queries

T-SQL is a set-based language. This means that it does not go through records row-by-row, but instead pulls data from the server one table at a time, or a subset of the table if it is filtered. This makes SQL Server very efficient to deal with large volumes of data, although writing a seemingly straightforward query to add five percent to the preceding row is quite complicated. SQL Server does not typically consider what row a record is on, it looks at the data within that row.

- Set-based language
- Scripts
- Batches

```
CREATE TABLE HumanResources.Employees(
        EmployeeID int PRIMARY KEY,
        LastName nvarchar(25), FirstName nvarchar(25) );
GO
INSERT INTO HumanResources.Employees(
        EmployeeID, LastName, FirstName)
VALUES
        (123, O'Neill, Carlene);
GO
```

T-SQL scripts are stored in script files with an **.sql** extension. These can be further organized into projects.

Inside each file, the script can be ordered into batches, that are marked with the word **GO** at the end. It is important to realize that each batch is run in its entirety before the next one is started. This is important if things need to happen in a certain order. For example, if you had a script that created a table, and then populated it with data, it would fail without batches. SQL Server would analyze the batch and reject the statement that populated the table because the table does not currently exist. If you write the script to create the table, type **GO**, and then write the script to populate the table. It will succeed because the table exists when SQL Server assesses the second batch.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Sequencing Activity

Put the following T-SQL commands in order by numbering each to create a script that will execute without errors.

	Steps
	GO
	GO
	CREATE TABLE HumanResources.Employees(EmployeeID int PRIMARY KEY, LastName nvarchar(25), FirstName nvarchar(25));
	INSERT INTO HumanResources.Employees(EmployeeID, LastName, FirstName) VALUES (123, O'Neill, Carlene);

Lesson 2
SQL Server Editions and Versions

In this lesson, you will learn about the editions and versions of Microsoft SQL Server. You will learn which editions of SQL Server 2014 are available, their distinguishing features, and which editions would be best to use when planning a new deployment.

Lesson Objectives

After completing this lesson, you will be able to describe:

- The versions of SQL Server.
- The editions of SQL Server 2014.

SQL Server Versions

SQL Server 2014 is the latest version in the history of SQL Server development. Originally developed for the OS/2 operating system (versions 1.0 and 1.1), SQL Server versions 4.21 and later moved to the Windows® operating system.

SQL Server's engine received a major rewrite for version 7.0, and subsequent versions have continued to improve and extend SQL Server's capabilities, from the workgroup to the largest enterprises.

SQL Server Versions

Version	Release Year
1.0	1989
1.1	1991
4.2	1992
4.21	1994
6.0	1995
6.5	1996
7.0	1998
2000	2000
2005	2005
2008	2008
2008 R2	2010
2012	2012
2014	2014

Note: Although its name might suggest it, SQL Server 2008 R2 is not a service pack for SQL Server 2008. It is an independent version (number 10.5) with enhanced multiserver management capabilities, as well as new business intelligence (BI) features.

Question: Have you worked with any versions of SQL Server prior to SQL Server 2012?

SQL Server Editions

SQL Server offers several editions providing different feature sets that target various business scenarios. In the SQL Server 2012 release, the number of editions was streamlined from previous versions. The main editions are:

- **Enterprise**, which is the flagship edition. It contains all of SQL Server 2014's features, including BI services and support for virtualization.

• SQL Server Editions

Main Editions	Other Editions
Enterprise	Parallel Data Warehouse
Standard	Web
Business Intelligence	Developer
	Express
	Express LocalDB

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

- **Standard**, which includes the core database engine, as well as core reporting and analytics capabilities. However, it supports fewer processor cores and does not offer all the availability, security, and data warehousing features found in Enterprise.
- **Business Intelligence**, which is a new edition. It provides the core database engine, full reporting and analytics capabilities, and full BI services. However, like the Standard edition, it supports fewer processor cores and does not offer all the availability, security, and data warehousing features.

SQL Server 2014 also offers other editions, such as Parallel Data Warehouse, Web, Developer, and Express, each targeted for specific use cases.

This course uses core database engine features found in all editions.

SQL Server does not have to run locally, but can also operate as a cloud-based database, taking two forms. SQL Server could be running on a cloud-based server that your organization has provisioned and integrated with your infrastructure. If this is the case, and the infrastructure is properly set up, you should treat it as an instance of SQL Server on your network. In fact, this might be the case and you are completely unaware of it.

The other alternative is the Microsoft Azure™ SQL Database. This allows you to provision databases that use SQL Server technology in the cloud, but without having to provision and configure a whole virtual machine.

Additional Reading: For more information on the use of T-SQL in Microsoft Azure SQL Databases, go to the MSDN article **Azure SQL Database Transact-SQL Information**: http://aka.ms/arl8x9

Check Your Knowledge

Question	
You are deploying a new customer relationship management system based on Microsoft SQL Server 2014. The board has decided that, because of the large volume of business-critical data this application will contain, advanced business intelligence features must be available for analysts. Which edition of SQL Server should you use for this system?	
Select the correct answer.	
	Standard Edition
	Enterprise Edition
	Business Intelligence Edition
	Either Enterprise Edition or Business Intelligence Edition
	Any edition is appropriate for these requirements

Lesson 3
Getting Started with SQL Server Management Studio

In this lesson, you will learn how to use SQL Server Management Studio (SSMS) to connect to an instance of SQL Server, explore the databases contained in the instance, and work with script files containing

T-SQL queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Start SSMS.
- Use SSMS to connect to on-premises SQL Server instances.
- Explore a SQL Server instance using Object Explorer.
- Create and organize script files.
- Execute T-SQL queries.
- Use Books Online.

Starting SSMS

SSMS is an integrated management, development, and querying application with many features for exploring and working with your databases. SSMS is based on the Visual Studio shell. If you have experience with Visual Studio, you will likely feel comfortable with SSMS.

- Launch SSMS from the Windows Start screen
  - Or type "SSMS" into the "Search Programs and Files" box
- Connect to an instance or work disconnected
- Settings include:
  - Fonts and colors, line numbering, word wrap
  - Which windows open at start
- Useful windows include query editor, Object Explorer, and Solution Explorer

To start SSMS, you may:

- Use its shortcut on the Windows Start screen.
- Enter its filename, SSMS.EXE, in a command prompt window.

By default, SSMS will display a Connect to Server dialog box you can use to specify the server (or instance) name and your security credentials. If you use the Options button to access the Connection Properties tab, you can also supply the database to which you wish to connect. However, you can explore many SSMS features without initially connecting to an SQL Server instance, so you may also cancel the Connect to Server box and link to a server later.

After SSMS is running, you may wish to explore some of its settings, such as those found in the Tools, Options box. SSMS can be customized in many ways, such as setting a default font, enabling line numbers for scripts, and controlling the behavior of its many windows.

For more information on using SQL Server Management Studio, go to Use SQL Server Management Studio in Books Online:

Use SQL Server Management Studio

http://go.microsoft.com/fwlink/?LinkID=402707

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY, STUDENT USE PROHIBITED

Connecting to SQL Server

To connect to an instance of SQL Server, you need to specify several items, no matter which tool you use:

- Connecting to SQL Server requires three items:
  - Instance name
    - Use the form host\instance, except for the default instance
  - Database name
    - A default database can be assigned to logons
  - Authentication
    - May be Windows Authentication or SQL Server Authentication
    - Account must be provisioned by a database administrator

- The name of the instance to which you want to connect in the form: hostname\instancename. For example, MIA-SQL\Proseware would connect to the Proseware instance on the Windows server named MIA-SQL. If you are connecting to the default instance, you may omit the instance name.
- The name of the database. If you do not specify this, you will be connected to the database designated as your account's default by the database administrator, or to the master database if no default has been specifically assigned.
- The authentication mechanism required by the server. This may be Windows Authentication, in which your Windows network credentials will be passed to SQL Server (no entry required), or SQL Server Authentication, in which a username and password for your account must be created by a database administrator (you enter them at connection time).

  Question: Which authentication method do you use to log on to SQL Server in your organization?

Working with Object Explorer

Object Explorer is a graphical tool for managing SQL Server instances and databases. It is one of several SSMS window panes available from the View menu. Object Explorer provides direct interaction with most SQL Server data objects, such as tables, views, and procedures. Right-clicking an object, such as a table, will display context-sensitive commands, including query and script generators for object definitions.

- Object Explorer is a graphical tool for managing SQL Server and browsing objects at instance and database levels
- Context-sensitive shortcut menus provide frequently used commands
- Object definitions can be scripted out as T-SQL statements to a query window, the clipboard, or a file
- New query windows can be started by right-clicking databases in Object Explorer
  - Changing the selected object in Object Explorer does not change the existing connection in the query window

Note: Any operation performed in SSMS requires appropriate permissions granted by a database administrator. Being able to see an object or command does not necessarily imply permission to use the object or issue the command.

SQL Server query writers most commonly use Object Explorer to learn about the structure and definition of the data objects they want to use in their queries. For example, to learn the names of columns in a table, you follow these steps:

1. Connect to the SQL Server, if necessary.
2. Expand the Databases folder to expose the list of databases.
3. Expand the relevant database to expose the Tables folder.

4. Expand the Tables folder to view the list of tables in the database.
5. Locate the table you are interested in and expand it to find the Columns folder. The Columns folder will display the names, data types, and other information about the columns that make up the table. You can even drag the name of a database, table, or column into the query window to have it entered and avoid typing it yourself.

Note: Selecting objects in the Object Explorer pane does not change any connections made in other windows.

Working with Script Files and Projects

SSMS allows you to create and save T-SQL code in text files, typically given an .sql file extension. Like other Windows applications that open, edit, and save files, SSMS provides access to file management through the File menu, as well as standard toolbar buttons.

- T-SQL scripts are text files, typically given an .sql extension
- SSMS can open, edit, and execute code in script files directly
- SSMS can also organize scripts into logical containers.
  - Scripts can be associated with SQL Server Script projects (*.ssmssqlproj)
  - Projects can be organized in solutions (*.ssmssln)
- Opening a solution in SSMS provides convenient access to scripts in projects
- You will use projects throughout this course and its labs

In addition to directly manipulating individual script files, SSMS provides a mechanism for initially saving groups of files together and for opening, saving, and closing them together. This mechanism uses several conceptual layers to work with T-SQL script files and related documents, using the Solution Explorer pane to display and control them:

Object	Parent	Description
Solution	None	Top-level container for projects. Stored as a text file with an .ssmssln extension, which references components contained within it. May contain multiple projects. Displayed at the top of the object hierarchy in Solution Explorer.
Project	Solution	Container for T-SQL scripts (called queries), stored database connection metadata, and miscellaneous files. Stored as a text file with an .ssmssqlproj extension, which references component scripts and other files.
Script	Project	T-SQL script file with an .sql extension. The core item of work in SSMS.

The benefits of using scripts organized in projects and solutions include the ability to immediately open multiple script files in SSMS. You can open the solution or project file from within SSMS or Windows Explorer.

To create a new solution, click the File menu and click New Project. (There is no "New Solution" command.) Specify a name for the initial project, its parent solution, and whether you want the project to be stored in a subfolder below the solution file in the location you indicate. Click OK to create the parent objects.

To interact with Solution Explorer, open the pane (if necessary) from the View menu. To create a new script that will be stored as part of the project, right-click the Queries folder in the project and click New Query.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Note: Using the New Query toolbar button or the new query commands on the File menu will create a new script temporarily stored with the solution in the Miscellaneous Files folder. If you wish to move an existing open query document into a solution currently open in Solution Explorer, you will need to save the file. You can then drag the query into the project tree to save it in the Queries folder. This will make a copy of the script file and place it in the solution.

It is important to remember to save the solution when exiting SSMS or opening another solution to preserve changes to the file inventory. Saving a script using the Save toolbar button or the Save <Queryname>.sql command on the File menu will only save changes to the current script file contents. To save the entire solution and all its files, use the Save All command on the File menu or when prompted to save the .ssmssln and .ssmssqlproj files on exit.

Executing Queries

To execute T-SQL code in SSMS, you first need to open the .sql file that contains the queries, or type your query into a new query window. Then decide how much of the code in the script is to be executed as follows:

- Select the code you wish to execute.
- If nothing is selected, SSMS will execute the entire script, which is the default behavior.

When you have decided what you wish to execute, you can run the code by doing one of the following:

- Clicking the Execute button on the SSMS toolbar.
- Clicking the Query menu, then clicking Execute.
- Pressing the F5 key, the Alt+X keyboard shortcut, or the Ctrl+E keyboard shortcut.

By default, SSMS will display your results in a new pane of the query window. The location and appearance of the results can be changed in the Options box, accessible from the Tools menu. To toggle the results display and return to a full-screen T-SQL editor, use the Ctrl+R keyboard shortcut.

SSMS provides several formats for the display of query results:

- **Grid**, which is a spreadsheet-like view of the results, with row numbers and columns you can resize. You can use Ctrl+D to select this before executing a query.
- **Text**, which is a Windows Notepad-like display that pads column widths. You can use Ctrl+T to select this before executing a query.
- **File**, which allows you to directly save query results to a text file with an .rpt extension. Executing the query will prompt a results file location. The file may then be opened by any application that can read text files, such as Windows Notepad and SSMS. You can use Ctrl+Shift+F to select this before executing a query.

Using Books Online

Books Online (often abbreviated BOL) is the product documentation for SQL Server. BOL includes helpful information on SQL Server's architecture and concepts, as well as syntax reference for T-SQL. BOL can be accessed from the Help menu in SSMS. In a script window, context-sensitive help for T-SQL keywords is available by selecting the keyword and pressing Shift+F1.

- Books Online is the product documentation for SQL Server
- In SQL Server 2014, Books Online does not ship with the SQL Server installation media
- Configure Help to display content from MSDN Library or download Help Collections to a local computer
- Once configured, Help is available from:
  - SSMS query editor (context-sensitive)
  - SSMS Help menu
  - Windows Start menu

Books Online can be browsed directly on Microsoft's website:

Books Online for SQL Server 2014

http://go.microsoft.com/fwlink/?LinkID=402708

Note: Before SQL Server 2014, SQL Server provided the option to install Books Online locally during SQL Server setup. In SQL Server 2014, Books Online does not ship with the product installation media, so must be downloaded and installed separately.

The first time Help is invoked, you will be prompted to specify whether you wish to view Books Online content online or locally.

For detailed instructions on how to download, install, and configure Books Online for local offline use, go to the topic Get Started with Product Documentation for SQL Server:

Get Started with Product Documentation for SQL Server

http://go.microsoft.com/fwlink/?LinkID=402709

Demonstration: Introducing Microsoft SQL Server 2014

In this demonstration, you will see how to:

- Use SSMS to connect to an on-premises instance of SQL Server.
- Explore databases and other objects.
- Work with T-SQL scripts.

Demonstration Steps

Use SSMS to connect to an on-premises instance of SQL Server 2014

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod01\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.

Explore databases and other objects

1. If the **Object Explorer** pane is not visible, click **View** and click **Object Explorer**.
2. Expand the **Databases** folder to see the list of databases.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

3. Expand the **AdventureWorks** database.
4. Expand the **Tables** folder.
5. Expand the **Sales.Customer** table.
6. Expand the **Columns** folder.
7. Show the list of columns, and point out the data type information for the **ModifiedDate** column.

Work with T-SQL scripts

1. If the **Solution Explorer** pane is not visible, click **View** and click **Solution Explorer**. Initially, it will be empty.
2. Click the **File** menu, click **New**, click **Project**.
3. In the **New Project** box, under **Installed Templates**, click **SQL Server Management Studio Projects**.
4. In the middle pane, click **SQL Server Scripts**.
5. In the **Name** box, type **Module 1 Demonstration**.
6. In the **Location** box, type or browse to **D:\Demofiles\Mod01**.
7. Point out the solution name, then click **OK**.
8. In the **Solution Explorer** pane, right-click **Queries**, then click **New Query**.
9. Type the following T-SQL code:

```
USE AdventureWorks;
GO
SELECT CustomerID, AccountNumber
FROM Sales.Customer;
```

10. Select the code and click **Execute** on the toolbar.
11. Point out the results pane.
12. Click **File**, and then click **Save All**.
13. Click **File**, and then click **Close Solution**.
14. Click **File**, click **Recent Projects and Solutions**, and then click **Module 1 Demonstration.ssmssln**.
15. Point out the **Solution Explorer** pane.
16. Close SQL Server Management Studio without saving any files.

Check Your Knowledge

Question
You have been asked to run some test queries against the company's scheduling database. Administrators have provided the name of the server where the database is hosted and the name of the database itself. Permissions to run the necessary queries have been granted to your Active Directory account. You are logged onto a client computer with this Active Directory account and have started SQL Server Management Studio. What other information do you need to connect to the database?
Select the correct answer.

Question	
	Your Active Directory account username
	Your Active Directory account password
	The name of the Login created for you in the SQL Server instance
	The name of the instance that hosts the database
	The name of the user created for you in the SQL Server database

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Working with SQL Server 2014 Tools

Scenario

The Adventure Works Cycles Bicycle Manufacturing Company has adopted SQL Server 2014 as its relational database management system of choice. You are an information worker who will be required to find and retrieve business data from several SQL Server databases. In this lab, you will begin to explore the new environment and become acquainted with the tools for querying SQL Server.

Objectives

After completing this lab, you will be able to:

- Use SQL Server Management Studio.
- Create and organize T-SQL scripts.
- Use SQL Server Books Online.

Estimated Time: 30 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Working with SQL Server Management Studio

Scenario

You have been tasked with writing queries for SQL Server. Initially, you would like to become familiar with the development environment and, therefore you have decided to explore SQL Server Management Studio and configure the editor for your use.

The main tasks for this exercise are as follows:

1. Open Microsoft SQL Server Management Studio

2. Configure the Editor Settings

▶ Task 1: Open Microsoft SQL Server Management Studio

1. Start SSMS, but do not connect to an instance of SQL Server.
2. Close the **Object Explorer** and **Solution Explorer** windows.
3. Using the **View** menu, show the **Object Explorer** and **Solution Explorer** windows in SSMS.

▶ Task 2: Configure the Editor Settings

1. On the **Tools** menu, choose **Options** to open the **Options** window in SSMS and change the font size to 14 for the text editor.
2. Change several additional settings in the **Options** window:
   - Disable IntelliSense.
   - Change the tab size to 6 spaces for T-SQL.
   - Enable the option to include column headers when copying the result from the grid. Look under **Query Results, SQL Server, Results to Grid** for the check box **Include column headers when copying or saving the results**.

Results: After this exercise, you should have opened SSMS and configured editor settings.

Exercise 2: Creating and Organizing T-SQL scripts

Scenario

Usually you will organize your T-SQL code in multiple query files inside one project. You will practice how to create a project and add different query files to it.

The main tasks for this exercise are as follows:

1. Create a Project
2. Add an Additional Query File
3. Reopen the Created Project

▶ Task 1: Create a Project

1. Create a new project called MyFirstProject and store it in the folder D:\Labfiles\Lab01\Starter.
2. Add a new query file to the created project and name it MyFirstQueryFile.sql.
3. Save the project and the query file by clicking the **Save All** option.

▶ Task 2: Add an Additional Query File

1. Add an additional query file called MySecondQueryFile.sql to the created project and save it.
2. Open Windows Explorer, navigate to the project folder, and observe the created files in the file system.
3. Back in SSMS, using Solution Explorer, remove the query file MySecondQueryFile.sql from the created project. (Choose the **Remove** option, not **Delete**.)
4. Again, look in the file system. Is the file MySecondQueryFile.sql still there?
5. Back in SSMS, remove the file MyFirstQueryFile.sql and choose the **Delete** option. Observe the files in Windows Explorer. What is different this time?

▶ Task 3: Reopen the Created Project

1. Save the project, close SSMS, reopen SSMS, and open the project MyFirstProject.
2. Drag the query file MySecondQueryFile.sql from Windows Explorer to the Queries folder under the project MyFirstProject in Solution Explorer. (Note: If the **Solution Explorer** window is not visible, enable it as you did in exercise 1).
3. Save the project.

Results: After this exercise, you should have a basic understanding of how to create a project in SSMS and add query files to it.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Using Books Online

Scenario

To be effective in your upcoming training and exercises, you will practice how to use Books Online to efficiently check for T-SQL syntax.

The main tasks for this exercise are as follows:

1. Launch Books Online

2. Use Books Online

▶ Task 1: Launch Books Online

1. Launch Manage Help Settings from the Windows **Start** screen.

2. Configure Books Online to use the online option, not local help.

▶ Task 2: Use Books Online

1. Use Books Online to find information about SQL Server 2014 tools and add-in components.

Results: After this exercise, you should have a basic understanding of how to find information in Books Online.

Module Review and Takeaways

Review Question(s)

Question: Can an SQL Server database be stored across multiple instances?

Question: If no T-SQL code is selected in a script window, which lines will be run when you click the Execute button?

Question: What does an SQL Server Management Studio solution contain?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 2

Introduction to T-SQL Querying

Contents:

Module Overview **2-1**

Lesson 1: Introducing T-SQL **2-2**

Lesson 2: Understanding Sets **2-12**

Lesson 3: Understanding Predicate Logic **2-15**

Lesson 4: Understanding the Logical Order of Operations in SELECT Statements **2-17**

Lab: Introduction to T-SQL Querying **2-22**

Module Review and Takeaways **2-25**

Module Overview

Transact-SQL, or T-SQL, is the language in which you will write queries for Microsoft SQL Server 2014. In this module, you will learn that T-SQL has many elements in common with other computer languages, such as commands, variables, functions, and operators. You will also learn that T-SQL contains some unique elements that may require adjustment if your background includes experience with procedural languages. To make the most of your effort in writing T-SQL queries, you will also learn the process by which SQL Server evaluates your queries. Understanding the logical order of operations of SELECT statements will be vital to learning how to write effective queries.

Objectives

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

- The elements of T-SQL and their role in writing queries.
- The use of sets in SQL Server.
- The use of predicate logic in SQL Server.
- The logical order of operations in SELECT statements.

Lesson 1
Introducing T-SQL

In this lesson, you will learn the role of T-SQL in writing SELECT statements. You will learn about many of the T-SQL language elements and which ones will be useful for writing queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe Microsoft's implementation of the standard SQL language.
- Categorize SQL statements into their dialects.
- Identify the elements of T-SQL, including predicates, operators, expressions, and comments.

About T-SQL

T-SQL is Microsoft's implementation of the industry standard Structured Query Language. Originally developed to support the new relational data model at International Business Machines (IBM) in the early 1970s, SQL has become widely adopted in the industry. SQL became a standard of the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) in the 1980s.

- Structured Query Language (SQL)
  - Developed by IBM in 1970s
  - Adopted as a standard by ANSI and ISO standards bodies
  - Widely used in industry
    - PL/SQL, SQL Procedural Language, Transact-SQL
- Microsoft's implementation is Transact-SQL
  - Referred to as T-SQL
  - Query language for SQL Server 2012
- SQL is declarative, not procedural
  - Describe what you want, don't specify steps

The ANSI standard has gone through several revisions, including SQL-89 and SQL-92, whose specifications are either fully or partly supported by T-SQL. SQL Server 2014 also implements features from later standards, such as ANSI SQL-2008. Microsoft, like many vendors, has also extended the language to include SQL Server-specific features and functions.

Besides Microsoft's implementation as T-SQL in SQL Server, Oracle has PL/SQL, IBM has SQL PL, and Sybase maintains its own T-SQL operation.

An important concept to understand when working with T-SQL is that it is a set-based and declarative language, not a procedural one. When you write a query to retrieve data from SQL Server, you describe the data you wish to display, you do not tell SQL Server exactly how to retrieve it. Instead of providing a procedural list of steps to take, you provide the attributes of the data you seek.

For example, if you want to retrieve a list of customers who are located in Portland, a procedural method might look like this:

Procedural Method

```
Open a cursor to consume rows, one at a time.
Fetch the first cursor record.
Examine first row.
If the city is Portland, return the row.
Move to next row.
If the city is Portland, return the row.
Fetch the next record.
(Repeat until end of table is reached).
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Your procedural code must not only contain the logic to select the data that meets your needs, but you must also determine and execute a well-performing path through it.

Note: This course mentions cursors for comparative purposes, but does not provide training on writing code with them.

Go to Books Online (BOL) for definitions and concerns regarding the use of cursors:

Cursors

http://go.microsoft.com/fwlink/?LinkID=402710

With a declarative language such as T-SQL, you will provide the attributes and values that describe the set you wish to retrieve.

For example, see the following pseudo-code:

Declarative Language

```
Display all customers whose city is Portland.
```

With T-SQL, the SQL Server 2014 database engine will determine the optimal path to access the data and return a matching set. Your role is to learn to write efficient and accurate T-SQL code so you can properly describe the set you wish to retrieve.

If you have a background in other programming environments, adopting a new mindset may present a challenge. This course has been designed to help you bridge the gap between procedural and set-based declarative T-SQL.

Note: Sets will be discussed later in this module.

Categories of T-SQL Statements

T-SQL statements can be organized into several categories:

- **Data Manipulation Language**, or DML, is the set of T-SQL statements that focuses on querying and modifying data. This includes SELECT, the primary focus of this course, as well as modification statements such as INSERT, UPDATE, and DELETE. You will learn about SELECT statements throughout this course.
- **Data Definition Language**, or DDL, is the set of T-SQL statements that handles the definition and life cycle of database objects, such as tables, views, and procedures. This includes statements such as CREATE, ALTER, and DROP.
- **Data Control Language**, or DCL, is the set of T-SQL statements used to manage security permissions for users and objects. DCL includes statements such as GRANT, REVOKE, and DENY.

Note: DCL commands are beyond the scope of this course. For more information about SQL Server 2014 security, including DCL, see Microsoft Official Course 20462D: *Administering a Microsoft SQL Server Database*.

Additional information on DML, DDL, and DCL commands can be found in Books Online at:

Transact-SQL Reference (Database Engine)

http://go.microsoft.com/fwlink/?LinkID=402711

T-SQL Language Elements

Like many programming languages, T-SQL contains elements that you will use in queries. You will use predicates to filter rows, operators to perform comparisons, functions and expressions to manipulate data or retrieve system information, and comments to document your code. If you need to go beyond writing SELECT statements to create stored procedures, triggers, and other objects, you may use elements such as control-of-flow statements, variables to temporarily store values, and batch separators. The next several topics in this lesson will introduce you to many of these elements.

- Predicates and Operators
- Functions
- Variables
- Expressions
- Batch Separators
- Control of Flow
- Comments

Note: The purpose of this lesson is to introduce many elements of the T-SQL language, which will be presented here at a high conceptual level. Subsequent modules in this course will provide more detailed explanations.

T-SQL Language Elements: Predicates and Operators

The T-SQL language provides elements for specifying and evaluating logical expressions. In SELECT statements, you can use logical expressions to define filters for WHERE and HAVING clauses. You will write these expressions using predicates and operators.

Elements:	Predicates and Operators:
Predicates	IN, BETWEEN, LIKE
Comparison Operators	=, >, <, >=, <=, <>, !=, !>, !<
Logical Operators	AND, OR, NOT
Arithmetic Operators	+, -, *, /, %
Concatenation	+

T-SQL enforces operator precedence

Predicates supported by T-SQL include the following:

- IN, used to determine whether a value matches any value in a list or subquery.
- BETWEEN, used to specify a range of values.
- LIKE, used to match characters against a pattern.

Operators include several common categories:

- Comparison for equality and inequality tests: =, <, >, >=, <=, !=, !>, !< (Note that !>, !< and != are not ISO standard. It is best practice to use standard options when they exist).

MCT USE ONLY STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

- Logical, for testing the validity of a condition: AND, OR, NOT
- Arithmetic, for performing mathematical operations: +, -, *, /, % (modulo)
- Concatenation, for combining character strings: +
- Assignment, for setting a value: =

As with other mathematical environments, operators are subject to rules governing precedence. The following table describes the order in which T-SQL operators are evaluated:

Order of Evaluation	Operators
1	( ) Parentheses
2	*, /, % (Multiply, Divide, Modulo)
3	+, - (Add/Positive/Concatenate, Subtract/Negative)
4	=, <, >, >=, <=, !=, !>, !< (Comparison)
5	NOT
6	AND
7	BETWEEN, IN, LIKE, OR
8	= (Assignment)

Information on other categories of operators, including bitwise, unary, and scope assignment, can be found in Books Online at:

Operators (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402714

T-SQL Language Elements: Functions

SQL Server 2014 provides a wide variety of functions available to your T-SQL queries. They range from scalar functions such as SYSDATETIME, which return a single-valued result, to others that operate on and return entire sets, such as the windowing functions you will learn about later in this course.

As with operators, SQL Server functions can be organized into categories. Here are some common categories of scalar (single-value) functions available to you for writing queries:

- String functions
  - SUBSTRING, LEFT, RIGHT, LEN, DATALENGTH
  - REPLACE, REPLICATE
  - UPPER, LOWER, RTRIM, LTRIM

- Date and time functions
  - GETDATE, SYSDATETIME, GETUTCDATE
  - DATEADD, DATEDIFF
  - YEAR, MONTH, DAY
- Aggregate functions
  - SUM, MIN, MAX, AVG
  - COUNT, COUNTBIG
- Mathematical functions
  - RAND, ROUND, POWER, ABS
  - CEILING, FLOOR

Note: The purpose of this lesson is to introduce many elements of the T-SQL language, which will be presented here at a high conceptual level. Subsequent modules in this course will provide more detailed explanations.

Additional information on these functions, with sample code, can be found in Books Online at:

Built-in Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402712

T-SQL Language Elements: Variables

Like many programming languages, T-SQL provides a means of temporarily storing a value of a specific data type. However, unlike other programming environments, all user-created variables are local to the T-SQL batch that created them, and are visible only to that batch. There are no global or public variables available to SQL Server users.

To create a local variable in T-SQL, you must provide a name, data type, and initial value. The name must start with a single @ (at) symbol, and the data type must be system-supplied or user-defined and stored in the database your code will run against.

- Local variables in T-SQL temporarily store a value of a specific data type
- Name begins with single @ sign
  - @@ reserved for system functions
- Assigned a data type
- Must be declared and used within the same batch
- In SQL Server 2008 and later, you can declare and initialize in the same statement

```
DECLARE @MyVar int = 30;
```

Note: You may find references in SQL Server literature, websites, and so on, to so-called "system variables," named with a double @@, such as @@ERROR. It is more accurate to refer to these as system functions, since users may not assign a value to them. This course will differentiate user variables prefixed with a single @ from system functions prefixed with @@.

If your variable is not initialized in the DECLARE statement, it will be created with a value of NULL and you can subsequently assign a value with the SET statement. SQL Server 2008 introduced the capability to name and initialize a variable in the same statement.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The following example creates a local integer variable called MyVar and assigns it an initial value of 30:

Integer Variable

```
DECLARE @MyVar int = 30;
```

The following example creates a local date variable called MyDate and separately assigns it an initial value of 15 Feb 2012:

Date Variable

```
DECLARE @MyDate date;
SET @MyDate = '20120215';
```

You will learn more about data types, including dates, and about T-SQL variables, later in this course.

If persistent storage or global visibility for a value is needed, consider creating a table in a database for that purpose. SQL Server provides both session-temporary and permanent storage in databases. For more information on temporary tables and objects, go to:

Special Table Types

http://go.microsoft.com/fwlink/?LinkID=402715

T-SQL Language Elements: Expressions

T-SQL provides the use of combinations of identifiers, symbols, and operators that are evaluated by SQL Server to return a single result. These combinations are known as expressions, providing a useful and powerful tool for your queries. In SELECT statements, you may use expressions:

- Combination of identifiers, values, and operators evaluated to obtain a single result
- Can be used in SELECT statements
  - SELECT clause
  - WHERE clause
- Can be single constant, single-valued function, or variable
- Can be combined if expressions have same the data type

```
SELECT YEAR(orderdate) + 1 ...
SELECT qty * unitprice ...
```

- In the SELECT clause to operate on and/or manipulate columns.
- As CASE expressions to replace values matching a logical expression with another value.
- In the WHERE clause to construct predicates for filtering rows.
- As table expressions to create temporary sets used for further processing.

Note: The purpose of this lesson is to introduce many elements of the T-SQL language, which will be presented here at a high conceptual level. Subsequent modules in this course will provide more detailed explanations.

Expressions may be based on a scalar (single-value) function, on a constant value, or on variables. Multiple expressions may be joined using operators if they have the same data type or if the data type can be converted from a lower precedence to a higher precedence (for example, int to money).

The following example of an expression operates on a column to add an integer to the results of the YEAR function on a datetime column:

Expression

```
SELECT YEAR(orderdate) AS currentyear, YEAR(orderdate) + 1 AS nextyear
FROM Sales.Orders;
```

Note: The preceding example uses T-SQL techniques, such as column aliases and date functions, which will be covered later in this course.

T-SQL Language Elements: Control of Flow, Errors, and Transactions

While T-SQL is primarily a data retrieval language and not a procedural language, it does support a limited set of statements that provide some control of flow during execution.

Some of the commonly-used control-of-flow statements include:

- IF . . . ELSE, for providing branching control based on a logical test.
- WHILE, for repeating a statement or block of statements while a condition is true.
- BEGIN . . . END, for defining the extents of a block of T-SQL statements.
- TRY . . . CATCH, for defining structure exception handling (error handling).
- BEGIN TRANSACTION, for marking a block of statements as part of an explicit transaction. Ended by COMMIT TRANSACTION or ROLLBACK TRANSACTION.

Note: Control-of-flow operators are not used in stand-alone queries. If your primary role is as a report writer, for example, it is unlikely that you will need to use them. However, if your responsibilities include creating objects such as stored procedures and triggers, you will find these elements useful.

T-SQL Language Elements: Comments

T-SQL provides two mechanisms for documenting code or instructing the database engine to ignore certain statements. Which method you use will typically depend on the number of lines of code you want ignored:

- For single lines, or very few lines of code, use the -- (double dash) to precede the text to be marked as a comment. Any text following the dashes will be ignored by SQL Server.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

- For longer blocks of code, enclose the text between /* and */ characters. Any code between the characters will be ignored by SQL Server.

The following example uses the -- (double dash) method to mark comments:

```
-- This entire line of text will be ignored.
DECLARE @MyVar int = 30; --only the text following the dashes will be ignored.
```

The following example uses the /* comment block */ method to mark comments:

```
*/
  This is comment text that will be ignored.
*/
```

Many query editing tools, such as SSMS or SQLCMD, will color-code commented text in a different color than the surrounding T-SQL code. In SSMS, use the Tools, Options dialog box to customize the colors and fonts in the T-SQL script editor.

T-SQL Language Elements: Batch Separators

SQL Server client tools, such as SSMS, send commands to the database engine in sets called batches. If you are manually executing code, such as in a query editor, you can choose whether to send all the text in a script as one batch. You may also choose to insert separators between certain sections of code.

- Batches are sets of commands sent to SQL Server as a unit
- Batches determine variable scope, name resolution
- To separate statements into batches, use a separator:
  - SQL Server tools use the GO keyword
  - GO is not an SQL Server T-SQL command!
  - GO [count] times feature added in SQL Server 2005

The specification of a batch separator is handled by your client tool. For example, the keyword GO is the default batch separator in SSMS. You can change this for the current query in Query | Query Options or globally in Tools | Options | Query Execution.

For most simple query purposes, batch separators are not used, as you will be submitting a single query at a time. However, when you need to create and manipulate objects, you may need to separate statements into distinct batches. For example, a CREATE VIEW statement may not be included in the same batch as other statements.

The following is an example of a CREATE TABLE and CREATE VIEW statement in the same batch:

Code That Requires Multiple Batches

```
CREATE TABLE t1 (col1 int);
CREATE VIEW v1 as SELECT * FROM t1;
```

The previous example returns the following error:

```
Msg 111, Level 15, State 1, Line 2
'CREATE VIEW' must be the first statement in a query batch.
```

Note that user-declared variables are considered local to the batch in which they are declared. If a variable is declared in one batch and referenced in another, the second batch would fail.

For example, the following statements sent together as one batch work properly:

Local Variable

```
DECLARE @cust int = 5;

SELECT custid, companyname, contactname
FROM Sales.Customers
WHERE custid = @custid;
```

However, if a batch separator was inserted between the variable declaration and the query in which the variable is used, an error would occur.

The following example separates the variable declaration from its use in a query:

Variable Separated by Batch

```
DECLARE @cust int = 5;
GO
SELECT custid, companyname, contactname
FROM Sales.Customers
WHERE custid = @custid;
```

The previous example returns the following error:

```
Msg 137, Level 15, State 2, Line 3
Must declare the scalar variable "@custid".
```

Note: Variables will be covered in more depth later in this course.

Demonstration: T-SQL Language Elements

In this demonstration, you will see how to:

- Use T-SQL language elements.

Note that some elements will be covered in more depth in later modules.

Demonstration Steps

Use T-SQL Language Elements

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod02\Setup.cmd as administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance.
4. Open the Demo.ssmssln solution in the D:\Demofiles\Mod02\Demo folder.
5. On the **View** menu, click **Solution Explorer**.
6. Open the 11 – Demonstration A.sql script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Check Your Knowledge

Question	
From the following T-SQL elements, select the one that contains an expression:	
Select the correct answer.	
	SELECT FirstName, LastName, SkillName AS Skill
	FROM HumanResources.Employees
	JOIN HumanResources.Skills ON Employees.ID = Skills.EmployeeID
	WHERE Skill.Level + Skill.Confidence > 10

Lesson 2
Understanding Sets

The purpose of this lesson is to introduce the concepts of the set theory, one of the mathematical underpinnings of relational databases, and to help you apply it to how you think about querying SQL Server.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the role of sets in a relational database.
- Understand the impact of sets on your T-SQL queries.
- Describe attributes of sets that may require special treatment in your queries.

Set Theory and SQL Server

The set theory is one of the mathematical foundations of the relational model and so is foundational for working with SQL Server 2014. While you might be able to make progress writing queries in T-SQL without an appreciation of sets, you may eventually have difficulty expressing some of them in a single, well-performing statement.

- Set theory is a mathematical basis for the relational database model
- What is a set?
  - A collection of distinct objects considered as a whole
  - "All customers who live in Portland"

Characteristics of Set Element	Example
Elements of a set called members	Customer as member of set Customers
Elements of a set are described by attributes	Customer first name, last name, birthdate
Elements must be distinct, or unique	Customer ID

This lesson will set the stage for you to begin "thinking in sets" and understanding their nature. In turn, this will make it easier for you to:

- Take advantage of set-based statements in T-SQL.
- Understand why you still need to sort your query output.
- Understand why a set-based, declarative approach, rather than a procedural one, works best with SQL Server 2014.

For our purposes, without delving into the mathematics supporting the set theory, we can define a set as "a collection of definite, distinct objects considered as a whole." In terms applied to SQL Server databases, we can think of a set as a single unit (such as a table) that contains zero or more members of the same type. For example, a Customers table represents a set, specifically the set of all customers. You will also see that the results of a SELECT statement also form a set, which will have important ramifications when learning about subqueries and table expressions, for example.

As you learn more about certain T-SQL query statements, it will be important to think of the entire set, instead of individual members, at all times. This will better equip you to write set-based code, instead of thinking one row at a time. Working with sets requires thinking in terms of operations that occur "all at once" instead of one at a time. This may require an adjustment for you, depending on your background.

After "collection," the next critical term in our definition is "distinct," or unique. All members of a set must be unique. In SQL Server, uniqueness is typically implemented using keys, such as a primary key column.

However, once you start working with subsets of data, it's important to be mindful of how you can uniquely address each member of a set.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

This brings us back to the consideration of the set as a "whole." Noted SQL language author Joe Celko suggests mentally adding the phrase "Set of all…" in front of the names of SQL objects that represent sets ("set of all customers," for example). This will help you remember that you are addressing a collection of elements when you write T-SQL code, not just one element at a time.

One important consideration is what is omitted from the set theory – any requirement regarding the order of elements in a set. In short, there is no predefined order in a set. Elements may be addressed (and retrieved) in any order. Applied to your queries, this means that if you need to return results in a certain order, you must use the ORDER BY clause in your SELECT statements. You will learn more about ORDER BY later in this course.

Additional Reading: More information on the set theory and its application to SQL Server queries can be found in Chapter 1 of Itzik Ben-Gan's *Inside Microsoft® SQL Server® 2008: T-SQL Querying* (Microsoft Press, 2009) and Chapter 2 of Itzik Ben-Gan's *Microsoft SQL Server 2008: T-SQL Fundamentals* (Microsoft Press, 2008). For more information on the use of "Set of all…" see Joe Celko's *Thinking in Sets* (Morgan Kaufman, 2008).

Set Theory Applied to SQL Server Queries

Given the set-based foundation of databases, there are a few considerations and recommendations to be aware of when writing efficient T-SQL queries:

Application of Set Theory	Comments
Act on all elements of a set at once	Query the whole table at once
Use declarative, set-based processing	Tell the engine what you want to retrieve
Avoid cursor-based processing	Do not tell the engine how to retrieve it. Do not process each result individually
Elements of set must be unique	Define unique keys in a table
No defined order to result set	Items may be returned in any order. This requires explicit sort instructions if an order is desired

- Act on the whole set at once. This translates to querying the whole table at once, instead of cursor-based or iterative processing.
- Use declarative, set-based processing. Tell SQL Server what you want to retrieve by describing its attributes, not by navigating to its position.
- Ensure that you are addressing elements via their unique identifiers, such as keys, when possible. For example, write JOIN clauses referencing unique keys on one side of the relationship.
- Provide your own sorting instructions because result sets are not guaranteed to be returned in any order.

Categorize Activity

Place each employee into the appropriate set. Indicate your answer by writing the set number to the right of each item.

Items	
1	Patrick Lorenzen Works in: London Skills: SharePoint Administration SQL Server Administration
2	Nickolas McLaughlin Works in: Seattle Skills: C# JavaScript

Items	
	SQL Server Administration
3	Naiyana Kunakorn Works in: Washington DC Skills: JavaScript T-SQL XML
4	Jeanie Sheppard Works in: Buenos Aires Skills: C# JavaScript T-SQL
5	Zachary Parsons Works in: Seattle Skills: Active Directory Administration SharePoint Administration SQL Server Administration
6	Carolos Lamy Works in: London Skills: JavaScript XML
7	Frederic Towle Works in: Tokyo Skills: Active Directory Administration T-SQL

Category 1		Category 2		Category 3
Employees in London		Employees who know T-SQL		Employees in Seattle who know SQL Server Administration

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Understanding Predicate Logic

Along with set theory, predicate logic is another mathematical foundation for the relational database model, and with it, SQL Server 2014. Unlike the set theory, you probably have a fair amount of experience with predicate logic, even if you have never used the term to describe it. This lesson will introduce predicate logic and examine its application to querying SQL Server.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the role of predicate logic in a relational database.
- Understand the use of predicate logic on your T-SQL queries.

Predicate Logic and SQL Server

- Predicate logic is a mathematical basis for the relational database model
- In theory, a predicate is a property or expression that is either true or false
- Predicate is also referred to as a Boolean expression

In theory, predicate logic is a framework for expressing logical tests that return true or false. A predicate is a property or expression that is true or false. You may have heard this referred to as a Boolean expression.

Taken by themselves, predicates make comparisons and express the results as true or false. However, in T-SQL, predicates don't stand alone. They are usually embedded in a statement that does something with the true or false result, such as a WHERE clause to filter rows, a CASE expression to match a value, or even a column constraint governing the range of acceptable values for that column in a table's definition.

There’s one important omission in the formal definition of a predicate—how to handle unknown, or missing, values. If a database is set up so that missing values are not permitted (through constraints, or default value assignments), then perhaps this is not an important omission. In most real-world environments, however, you will have to account for missing or unknown values, requiring you to extend your understanding of predicates from two possible outcomes (true or false) to three—true, false, or unknown.

The use of NULLs as a mark for missing data will be further discussed in the next topic, as well as later in this course.

Predicate Logic Applied to SQL Server Queries

As you have been learning, the ability to use predicates to express comparisons in terms of true, false, or unknown is vital to writing effective queries in SQL Server. Although we have been discussing them separately, predicates do not stand alone, syntactically speaking. You will typically use predicates in any of the following roles within your queries:

- Filtering data (in WHERE and HAVING clauses).
- Providing conditional logic to CASE expressions.
- Joining tables (in the ON filter).
- Defining subqueries (in EXISTS tests, for example).

Additionally, predicates have uses outside SELECT statements, such as in CHECK constraints to limit values permitted in a column, and in control-of-flow elements, such as an IF statement.

In mathematics, we only need to consider values that are present, so predicates can result only in true or false values (known in predicate logic as the law of the excluded middle). In databases, however, you will likely have to account for missing values, and the interaction of T-SQL predicates with missing values results in an unknown. Ensure that you have accounted for all three possible outcomes—true, false, or unknown—when you are designing your query logic. You will learn how to use three-valued logic in WHERE clauses later in this course.

Note: The purpose of this lesson is to introduce many elements of the T-SQL language, which will be presented here at a high conceptual level. Subsequent modules in this course will provide more detailed explanations.

Check Your Knowledge

Question	
From the following T-SQL elements, select the one that can include a predicate.	
Select the correct answer.	
	WHERE clauses
	JOIN conditions
	HAVING clauses
	IF statements
	All of the above

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 4

Understanding the Logical Order of Operations in SELECT Statements

T-SQL is unusual as a programming language in one key aspect. The order in which you write a statement is not necessarily that in which the database engine will evaluate and process it. Database engines may optimize their execution of a query, as long as the accuracy of the result (as determined by the logical order) is retained. As a result, unless you learn the logical order of operations, you may find both conceptual and practical obstacles to writing your queries. This lesson will introduce the elements of a SELECT statement, delineate the order in which the elements are evaluated, and then apply this understanding for a practical approach to writing queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the elements of a SELECT statement.
- Understand the order in which clauses in a SELECT statement are evaluated.
- Apply your understanding of the logical order of operations to writing SELECT statements.

Elements of a SELECT Statement

To accomplish our goal of understanding the logical order of operations, we need to look at a SELECT statement as a whole, including a number of optional elements. However, this lesson is not designed to provide detailed information about these elements. Each part of a SELECT statement will be discussed in subsequent modules. Understanding the details of a WHERE clause, for example, is not required in order to understand its place in the sequence of events.

Element	Expression	Role
SELECT	<select list>	Defines which columns to return
FROM	<table source>	Defines table(s) to query
WHERE	<search condition>	Filters rows using a predicate
GROUP BY	<group by list>	Arranges rows by groups
HAVING	<search condition>	Filters groups using a predicate
ORDER BY	<order by list>	Sorts the output

A SELECT statement is made up of mandatory and optional elements. Strictly speaking, SQL Server only requires a SELECT clause in order to execute without error. A SELECT clause without a FROM clause operates as though you were selecting from an imaginary table containing one row. You will see this behavior later in this course when you test variables. However, since a SELECT clause without a FROM clause cannot retrieve data from a table, we will treat stand-alone SELECT clauses as a special case that is not directly relevant to this lesson. Let's examine the elements, their high-level role in a SELECT statement, and the order in which they are evaluated by SQL Server.

Not all elements will be present in every SELECT query. However, when an element is present, it will always be evaluated in the same order with respect to the other ones present. For example, a WHERE clause will always be evaluated after the FROM clause and before a GROUP BY clause, if one exists.

Let's move on to discussing the order of these operations in the next topic.

Note: For the purposes of this lesson, additional optional elements such as DISTINCT, OVER, and TOP are omitted. They will be introduced and their order discussed in later modules.

Logical Query Processing

The order in which a SELECT statement is written is not that in which the SQL Server database engine evaluates and processes the statement.

Consider the following query:

Logical Query Processing

```
USE TSQL;
SELECT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid =71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
ORDER BY empid, orderyear;
```

Before we examine the run-time order of operations, let's briefly examine what the query does, although details on many clauses will need to wait until the appropriate module. The first line ensures we're connected to the correct database for the query. This line is not being examined for its run-time order.

If necessary, we need this to complete before the main SELECT query executes:

Change Connection

```
USE TSQL; -- change connection context to sample database.
```

The next line is the start of the SELECT statement as we wrote it, but as we'll see, it will not be the first line evaluated.

The SELECT clause returns the empid column and extracts just the year from the orderdate column:

Start of SELECT

```
SELECT empid, YEAR(orderdate) AS orderyear
```

The FROM clause identifies which table is the source of the rows for the query:

FROM Clause

```
FROM Sales.Orders
```

The WHERE clause filters the rows out of the Sales.Orders table, keeping only those that satisfy the predicate:

WHERE Clause

```
WHERE custid =71
```

The GROUP BY clause groups together the remaining rows by empid, and then by the year of the order:

GROUP BY Clause

```
GROUP BY empid, YEAR(orderdate)
```

After the groups are established, the HAVING clause filters them based on its predicate. Only employees with more than one sale per customer in a given year will pass this filter:

HAVING Clause

```
HAVING COUNT(*) > 1
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

The final clause, for the purposes of previewing this query, is the ORDER BY, which sorts the output by empid and then by year:

ORDER BY Clause

```
ORDER BY empid, orderyear;
```

Now that we've established what each clause does, let's look at the order in which SQL Server must evaluate them:

1. The FROM clause is evaluated first, to provide the source rows for the rest of the statement. Later in the course, we'll see how to join multiple tables together in a FROM clause. A virtual table is created and passed to the next step.
2. The WHERE clause is next to be evaluated, filtering those rows from the source table that match a predicate. The filtered virtual table is passed to the next step.
3. GROUP BY is next, organizing the rows in the virtual table according to unique values found in the GROUP BY list. A new virtual table is created, containing the list of groups, and is passed to the next step.

Note: From this point in the flow of operations, only columns in the GROUP BY list or aggregate functions may be referenced by other elements. This will have a significant impact on the SELECT list.

4. The HAVING clause is evaluated next, filtering out entire groups based on its predicate. The virtual table created in step 3 is filtered and passed to the next step.
5. The SELECT clause finally executes, determining which columns will appear in the query results.

Note: Because the SELECT clause is evaluated after the other steps, any column aliases created there cannot be used in clauses processed in steps 1 to 4.

In our example, the ORDER BY clause is the last to execute, sorting the rows as determined in its column list.

To apply this to our example query, here is the logical order at run time, with the USE statement omitted for clarity:

Logical Order

```
5.    SELECT empid, YEAR(orderdate) AS orderyear
1.    FROM Sales.Orders
2.    WHERE custid =71
3.    GROUP BY empid, YEAR(orderdate)
4.    HAVING COUNT(*) > 1
6.    ORDER BY empid, orderyear;
```

As we have seen, we do not write T-SQL queries in the same order in which they are logically evaluated. Since the runtime order of evaluation determines what data is available to clauses downstream from one another, it's important to understand the true logical order when writing your queries.

Applying the Logical Order of Operations to Writing SELECT Statements

Now you have learned the logical order of operations when a SELECT query is evaluated and processed, keep in mind the following considerations when writing a query. Note that some of these may refer to details you will learn in subsequent modules:

```
USE TSQL;

SELECT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
ORDER BY empid, orderyear;
```

- Decide which tables to query first, as well as any table aliases you will apply. This will determine the FROM clause.
- Decide which set or subset of rows will be retrieved from the table(s) in the FROM clause, and how you will express your predicate. This will determine your WHERE clause.
- If you intend to group rows, decide which columns will be grouped on. Remember that only columns in the GROUP BY clause, as well as aggregate functions such as COUNT, may ultimately be included in the SELECT clause.
- If you need to filter out groups, decide on your predicate and build a HAVING clause. The results of this phase become the input to the SELECT clause.
- If you are not using GROUP BY, determine which columns from the source table(s) you wish to display, and use any table aliases you created to refer to them. This will become the core of your SELECT clause. If you have used a GROUP BY clause, select from the columns in the GROUP BY clause, and add any additional aggregates to the SELECT list.
- Finally, remember that sets do not include any ordering. As a result, you will need to add an ORDER BY clause to guarantee a sort order if required.

Demonstration: Logical Query Processing

In this demonstration, you will see how to:

- View query output that illustrates logical processing order.

Demonstration Steps

View Query Output that Illustrates Logical Processing Order

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as ADVENTUREWORKS\Student with the password Pa$$w0rd, and run D:\Demofiles\Mod02\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod02\Demo folder.
3. On the **View** menu, click **Solution Explorer**. Then open the 21 – **Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Sequencing Activity

Put the following T-SQL elements in order by numbering each to indicate the order that SQL Server will process them in when they appear in a single SELECT statement.

	Steps
	SELECT
	GROUP BY
	WHERE
	HAVING
	ORDER BY
	FROM

Lab: Introduction to T-SQL Querying

Scenario

You are a business analyst for Adventure Work, who will be writing reports against corporate databases stored in SQL Server 2014. For you to become more comfortable with SQL Server querying, the Adventure Works IT department has provided you with a selection of common queries to run against their databases. You will review and execute these queries.

Objectives

After completing this lab, you will be able to:

- Execute basic SELECT statements.
- Execute queries that filter data.
- Execute queries that sort data.

Estimated Time: 30 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Executing Basic SELECT Statements

Scenario

The T-SQL script provided by the IT department includes a SELECT statement that retrieves all rows from the HR.Employees table, which includes the firstname, lastname, city, and country columns. You will execute the T-SQL script against the T-SQL database.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Execute the T-SQL Script
3. Execute a Part of the T-SQL Script

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab02\Starter folder as Administrator.

▶ Task 2: Execute the T-SQL Script

1. Using SSMS, connect to MIA-SQL using Windows authentication.
2. Open the project file D:\Labfiles\Lab02\Starter\Project\Project.ssmssln.
3. Open the T-SQL script 51 - Lab Exercise 1.sql.
4. Execute the script by clicking **Execute** on the toolbar (or press F5 on the keyboard). This will execute the whole script.
5. Observe the result and the database context.
6. Which database is selected in the **Available Databases** box?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Execute a Part of the T-SQL Script

1. Highlight the SELECT statement in the T-SQL script under the task 2 description and click Execute.
2. Observe the result. You should get the same result as in task 2.

Reader Aid: One way to highlight a portion of code is to hold down the Alt key while drawing a rectangle around it with your mouse. The code inside the drawn rectangle will be selected. Try it.

Results: After this exercise, you should know how to open the T-SQL script and execute the whole script or just a specific statement inside it.

Exercise 2: Executing Queries That Filter Data Using Predicates

Scenario

The next T-SQL script is very similar to the first one. The SELECT statement retrieves the same columns from the HR.Employees table, but uses a predicate in the WHERE clause to retrieve only rows with the value "USA" in the country column.

The main tasks for this exercise are as follows:

1. Execute the T-SQL Script
2. Apply Needed Changes and Execute the T-SQL Script
3. Uncomment the USE Statement

▶ Task 1: Execute the T-SQL Script

1. Close all open script files.
2. Open the project file D:\Labfiles\Lab02\Starter\Project\Project.ssmssln and the T-SQL script 61 - Lab Exercise 2.sql. Execute the whole script.
3. You get an error. What is the error message? Why do you think this happened?

▶ Task 2: Apply Needed Changes and Execute the T-SQL Script

1. Apply the needed changes to the script so that it will run without an error. (Hint: The SELECT statement is not the problem. Look at what is selected in the **Available Databases** box.) Test the changes by executing the whole script.
2. Observe the result. Notice that the result has fewer rows than the result in exercise 1, task 2.

▶ Task 3: Uncomment the USE Statement

1. Comments in T-SQL scripts can be written inside the line by specifying --. The text after the two hyphens will be ignored by SQL Server. You can also specify a comment as a block starting with /* and ending with */. The text in between is treated as a block comment and is ignored by SQL Server.
2. Uncomment the **USE TSQL**; statement.
3. Save and close the T-SQL script. Open the T-SQL script 61 - Lab Exercise 2.sql again. Execute the whole script.
4. Why did the script execute with no errors?
5. Observe the result and notice the database context in the **Available Databases** box.

Results: After this exercise, you should have a basic understanding of database context and how to change it.

Exercise 3: Executing Queries That Sort Data Using ORDER BY

Scenario

The last T-SQL script provided by the IT department has a comment: "This SELECT statement returns first name, last name, city, and country/region information for all employees from the USA, ordered by last name."

The main tasks for this exercise are as follows:

1. Execute the T-SQL Script
2. Uncomment the Needed T-SQL Statements and Execute Them

▶ Task 1: Execute the T-SQL Script

1. Open the project file D:\Labfiles\Lab02\Starter\Project\Project.ssmssln and the T-SQL script 71 - Lab Exercise 3.sql. Execute the whole script.
2. Observe the results. Why is the result window empty?

▶ Task 2: Uncomment the Needed T-SQL Statements and Execute Them

1. Observe that, before the USE statement, there are the characters -- which means that the USE statement is treated as a comment. There is also a block comment around the whole T-SQL SELECT statement. Uncomment both statements.
2. First execute the USE statement and then execute it, starting with the SELECT clause.
3. Observe the results. Notice that the results have the same rows as in exercise 1, task 2, but they are sorted by the lastname column.

Results: After this exercise, you should have an understanding of how comments can be specified inside T-SQL scripts.

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways

Review Question(s)

Question: Which category of T-SQL statements concerns querying and modifying data?

Question: What are some examples of aggregate functions supported by T-SQL?

Question: Which SELECT statement element will be processed before a WHERE clause?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 3

Writing SELECT Queries

Contents:

Module Overview	**3-1**
Lesson 1: Writing Simple SELECT Statements	**3-2**
Lesson 2: Eliminating Duplicates with DISTINCT	**3-6**
Lesson 3: Using Column and Table Aliases	**3-11**
Lesson 4: Writing Simple CASE Expressions	**3-16**
Lab: Writing Basic SELECT Statements	**3-19**
Module Review and Takeaways	**3-24**

Module Overview

The SELECT statement is used to query tables and views. You may also perform some manipulation of the data with SELECT before returning the results. It is likely that you will use the SELECT statement more than any other single statement in T-SQL. This module introduces you to the fundamentals of the SELECT statement, focusing on queries against a single table.

Objectives

After completing this module, you will be able to:

- Write simple SELECT statements.
- Eliminate duplicates using the DISTINCT clause.
- Use column and table aliases.
- Write simple CASE expressions.

Lesson 1
Writing Simple SELECT Statements

In this lesson, you will learn the structure and format of the SELECT statement, as well as enhancements that will add functionality and readability to your queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Understand the elements of the SELECT statement.
- Write simple single-table SELECT queries.
- Eliminate duplicate rows using the DISTINCT clause.
- Add calculated columns to a SELECT clause.

Elements of the SELECT Statement

The SELECT and FROM clauses are the primary focus of this module. You will learn more about the other clauses in later modules of this course. However, your understanding of the order of operations in logical query processing, from earlier in the course, will remain important for how you understand the proper way to write SELECT queries.

Clause	Expression
SELECT	<select list>
FROM	<table source>
WHERE	<search condition>
GROUP BY	<group by list>
ORDER BY	<order by list>

Remember that the FROM, WHERE, GROUP BY, and HAVING clauses will have been evaluated before the contents of the SELECT clause are processed. Therefore, elements you write in the SELECT clause, especially calculated columns and aliases, will not be visible to the other clauses.

Additional information on SELECT elements can be found at:

SELECT (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402716

MCT USE ONLY. STUDENT USE PROHIBITED

Retrieving Columns from a Table or View

The SELECT clause specifies the columns from the source table(s) or view(s) that you want to return as the result set of the query. In addition to columns from the source table, you may add others in the form of calculated expressions.

- Use SELECT with column list to display columns
- Use FROM to specify a source table or view
  - Specify both schema and table names
- Delimit names if necessary
- End all statements with a semicolon

Keyword	Expression
SELECT	<select list>
FROM	<table source>

```
SELECT companyname, country
FROM Sales.Customers;
```

The FROM clause specifies the name of the table or view that is the source of the columns in the SELECT clause. To avoid errors in name resolution, it is a best practice to specify both the schema and object name of the table, in the form SCHEMA.OBJECT, such as Sales.Customers.

If the table or view name contains irregular characters, such as spaces or other special characters, you will need to delimit, or enclose, the name. T-SQL supports the use of the ANSI standard double quotes "Sales Order Details" and the SQL Server-specific square brackets [Sales Order Details].

End all statements with a semicolon (;) character. In SQL Server 2014, semicolons are an optional terminator for most statements. However, future versions will require its use. For those current usages when a semicolon is required, such as some common table expressions (CTEs) and some Service Broker statements, the error messages returned for a missing semicolon are often cryptic. Therefore, you should adopt the practice of terminating all statements with a semicolon.

Displaying Columns

To display columns in a query, you need to create a comma-delimited column list. The order of the columns in your list will determine their display in the output, regardless of the order in which they are defined in the source table. This gives your queries the ability to absorb changes that others may make to the structure of the table, such as adding or reordering the columns.

- Displaying all columns
  - This is not a best practice in production code!

```
SELECT *
FROM Sales.Customers;
```

- Displaying only specified columns

```
SELECT companyname, country
FROM Sales.Customers;
```

T-SQL supports the use of the asterisk, or "star" character (*) to substitute for an explicit column list. This will retrieve all columns from the source table. While suitable for a quick test, avoid using the * in production work, as changes made to the table will cause the query to retrieve all current columns in the table's current defined order. This could cause bugs or other failures in reports or applications expecting a known number of columns returned in a defined order.

By using an explicit column list in your SELECT clause, you will always get the desired results, as long as the columns exist in the table. If a column is dropped, you will receive an error that will help identify the problem and fix your query.

MCT USE ONLY. STUDENT USE PROHIBITED

Using Calculations in the SELECT Clause

In addition to retrieving columns stored in the source table, a SELECT statement can perform calculations and manipulations. Calculations can manipulate the source column data and use built-in T-SQL functions, which you will learn about later in this course.

- Calculations are scalar, returning one value per row

Operator	Description
+	Add or concatenate
-	Subtract
*	Multiply
/	Divide
%	Modulo

- Using scalar expressions in the SELECT clause

```
SELECT unitprice, qty, (unitprice * qty)
FROM sales.orderdetails;
```

Since the results will appear in a new column, repeated once per row of the result set, calculated expressions in a SELECT clause must be scalar. In other words, they must return only a single value.

Calculated expressions may operate on other columns in the same row, on built-in functions, or a combination of the two:

Calculated Expression

```
SELECT unitprice, qty, (unitprice * qty)
FROM Sales.OrderDetails;
```

The results appear as follows:

```
unitprice             qty
--------------------- ------ ---------------------
14.00                 12     168.00
9.80                  10     98.00
34.80                 5      174.00
18.60                 9      167.40
```

Note that the new calculated column does not have a name returned with the results. To provide a name, you use a column alias, which you will learn about later in this module.

To use a built-in T-SQL function on a column in the SELECT list, you pass the name of the column to the function as an input:

Passing a Column

```
SELECT empid, lastname, hiredate, YEAR(hiredate)
FROM HR.Employees;
```

The results:

```
empid       lastname             hiredate
----------- -------------------- ----------------------- -------
1           Davis                2002-05-01 00:00:00.000 2002
2           Funk                 2002-08-14 00:00:00.000 2002
3           Lew                  2002-04-01 00:00:00.000 2002
```

You will learn more about date functions, as well as others, later in this course. The use of YEAR in this example is provided only to illustrate calculated columns.

Note: Not all calculations will be recalculated for each row. SQL Server may calculate a function's result once at the time of query execution, and reuse the value for each row. This will be discussed later in the course.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Writing Simple SELECT Statements

In this demonstration, you will see how to:

- Use simple SELECT queries

Demonstration Steps

Use Simple SELECT Queries

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod03\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod03\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

   Question: You have a table named Sales with the following columns: Country, NumberOfReps, TotalSales.

   You want to find out the average amount of sales a sales representative makes in each country. What SELECT query could you use?

Lesson 2
Eliminating Duplicates with DISTINCT

T-SQL queries may display duplicate rows, even if the source table has a key column enforcing uniqueness. Typically, this is the case when you retrieve only a few of the columns in a table. In this lesson, you will learn how to eliminate duplicates using the DISTINCT clause.

Lesson Objectives

In this lesson, you will learn to:

- Understand how T-SQL query results are not true sets and may include duplicates.
- Understand how DISTINCT may be used to remove duplicate rows from the SELECT results.
- Write SELECT DISTINCT clauses.

SQL Sets and Duplicate Rows

While the theory of relational databases calls for unique rows in a table, in practice T-SQL query results are not true sets. The rows retrieved by a query are not guaranteed to be unique, even when they come from a source table that uses a primary key to differentiate each row. Nor are the rows guaranteed to be returned in any particular order. You will learn how to address this with ORDER BY later in this course.

Add to this the fact that the default behavior of a SELECT statement is to include the keyword ALL, and you can begin to see why duplicate values might be returned by a query, especially when you include only some of the columns in a table (and omit the unique columns).

For example, consider a query that retrieves country names from the Sales.Customers table:

Select Query

```
SELECT country
FROM Sales.Customers;
```

A partial result shows many duplicate country names, which at best is too long to be easy to interpret. At worst, it gives a wrong answer to the question: "How many countries are represented among our customers?"

```
country
-------
Germany
Mexico
Mexico
UK
Sweden
Germany
Germany
France
UK
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
Austria
Brazil
Spain
France
Sweden
   ...
Germany
France
Finland
Poland
 (91 row(s) affected)
```

The reason for this output is that, by default, a SELECT clause contains a hidden default ALL statement:

ALL Statement

```
SELECT ALL country
FROM Sales.Customers;
```

In the absence of further instruction, the query will return one result for each row in the Sales.Customers table, but since only the country column is specified, you will see that column alone for all 91 rows.

Understanding DISTINCT

Replacing the default SELECT ALL clause with SELECT DISTINCT will filter out duplicates in the result set. SELECT DISTINCT specifies that the result set must contain only unique rows. However, it is important to understand that the DISTINCT option operates only on the set of columns returned by the SELECT clause. It does not take into account any other unique columns in the source table. DISTINCT also operates on all the columns in the SELECT list, not just the first one.

- Specifies that only unique rows can appear in the result set
- Removes duplicates based on column list results, not source table
- Provides uniqueness across set of selected columns
- Removes rows already operated on by WHERE, HAVING, and GROUP BY clauses
- Some queries may improve performance by filtering out duplicates prior to execution of SELECT clause

The logical order of operations also ensures that the DISTINCT operator will remove rows that may have already been processed by WHERE, HAVING, and GROUP BY clauses.

Continuing the previous example of countries from the Sales.Customers table, to eliminate the duplicate values, replace the silent ALL default with DISTINCT:

DISTINCT Statement

```
SELECT DISTINCT country
FROM Sales.Customers;
```

This will yield the desired results. Note that, while the results appear to be sorted, this is not guaranteed by SQL Server. The result set now contains only one instance of each unique output row:

```
Country
---------
Argentina
Austria
Belgium
Brazil
Canada
Denmark
```

MCT USE ONLY, STUDENT USE PROHIBITED

```
Finland
France
Germany
Ireland
Italy
Mexico
Norway
Poland
Portugal
Spain
Sweden
Switzerland
UK
USA
Venezuela
 (21 row(s) affected)
```

Note: You will learn additional methods for filtering out duplicate values later in this course. Once you have learned them, you may wish to consider the relative performance costs of filtering with SELECT DISTINCT versus those other means.

SELECT DISTINCT Syntax

Remember that DISTINCT looks at rows in the output set, created by the SELECT clause. Therefore, only unique combinations of column values will be returned by a SELECT DISTINCT clause.

For example, if you query a table with the following data in it, you might observe that there are only four unique first names and four unique last names:

```
SELECT DISTINCT <column list>
FROM <table or view>
```

```
SELECT DISTINCT companyname, country
FROM Sales.Customers;
```

```
companyname    country
-------------- -------
Customer AHPOP UK
Customer AHXHT Mexico
Customer AZJED Germany
Customer BSVAR France
Customer CCFIZ Poland
```

SELECT Statement

```
SELECT firstname, lastname
FROM Sales.Customers;
```

The results:

```
firstname  lastname
---------- --------------------
Sara       Davis
Don        Funk
Sara       Lew
Don        Davis
Judy       Lew
Judy       Funk
Yael       Peled
```

However, a SELECT DISTINCT query against both columns will retrieve all unique combinations of the two which, in this case, is the same seven employees.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

For a list of unique first names only, execute a SELECT DISTINCT only against the firstname column:

DISTINCT Syntax

```
SELECT DISTINCT firstname
FROM Sales.Customers;
```

The results:

```
firstname
----------
Don
Judy
Sara
Yael
 (4 row(s) affected)
```

A challenge in designing such queries is that, while you may need to retrieve a distinct list of values from one column, you might want to see additional attributes (columns) from others. Later in this course, you will see how to combine DISTINCT with the GROUP BY clause as a way of further processing and displaying information about distinct lists of values.

Demonstration: Eliminating Duplicates with DISTINCT

In this demonstration, you will see how to:

- Eliminate duplicate rows.

Demonstration Steps

Eliminate Duplicate Rows

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod03\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod03\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

   Question: You have company departments in five countries. You have the following query for the Human Resources database:

   ```
   SELECT DeptName, Country FROM HumanResources.Departments
   ```

   This returns:

   ```
   DeptName  Country
   --------- --------
   Sales     UK
   Sales     USA
   ```

```
Sales     France
Sales     Japan
Marketing USA
Marketing Japan
Research  USA
```

You add a DISTINCT keyword to the SELECT query. How many rows are returned

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Using Column and Table Aliases

When retrieving data from a table or view, a T-SQL query will name each column after its source. If desired, columns may be relabeled by the use of aliases in the SELECT clause. However, columns created with expressions will not be named automatically. Column aliases may be used to provide custom column headers. At the table level, aliases may be used in the FROM clause to provide a convenient way of referring to a table elsewhere in the query, enhancing readability.

Lesson Objectives

In this lesson you will learn how to:

- Use aliases to refer to columns in a SELECT list.
- Use aliases to refer to columns in a FROM clause.
- Understand the impact of the logical order of query processing on aliases.

Using Aliases to Refer to Columns

Column aliases can be used to relabel columns when returning the results of a query. For example, cryptic names of columns in a table such as qty may be replaced with quantity.

Expressions that are not based on a source column in the table will not have a name provided in the result set. This includes calculated expressions and function calls. While T-SQL doesn't require that a column in a result set have a name, it's a good idea to provide one.

In T-SQL, there are multiple methods of creating a column alias, with identical output results.

- Column aliases using AS

```
SELECT orderid, unitprice, qty AS quantity
FROM Sales.OrderDetails;
```

- Column aliases using =

```
SELECT orderid, unitprice, quantity = qty
FROM Sales.OrderDetails;
```

- Accidental column aliases

```
SELECT orderid, unitprice quantity
FROM Sales.OrderDetails;
```

One method is to use the AS keyword to separate the column or expression from the alias:

AS Keyword

```
SELECT orderid, unitprice, qty AS quantity
FROM Sales.OrderDetails;
```

Another method is to assign the alias before the column or expression, using the equals sign as the separator:

Alias With an Equals Sign

```
SELECT orderid, unitprice, quantity = qty
FROM Sales.OrderDetails;
```

Finally, you can simply assign the alias immediately following the column name, although this is not a recommended method:

Alias Following Column Name

```
SELECT orderid, unitprice, qty quantity
FROM Sales.OrderDetails;
```

While there is no difference in performance or execution, a variance in readability may cause you to choose one or the other as a convention.

Warning: Column aliases can also be accidentally created, by omitting a comma between two column names in the SELECT list.

For example, the following creates an alias for the **unitprice** column deceptively labeled **quantity**:

Accidental Alias

```
SELECT orderid, unitprice quantity
FROM Sales.OrderDetails;
```

The results:

```
orderid     quantity
----------- ---------------------
10248       14.00
10248       9.80
10248       34.80
10249       18.60
```

As you can see, this could be difficult to identify and fix in a client application. The only way to avoid this problem is to carefully list your columns, separating them properly with commas and adopting the AS style of aliases to make it easier to spot mistakes.

Question: Which style of column aliases do you prefer? Why?

Using Aliases to Refer to Tables

Aliases may also be used in the FROM clause to refer to a table, which can improve readability and save redundancy when referencing the table elsewhere in the query. While this module has focused on single-table queries, which don't necessarily benefit from table aliases, this technique will prove useful as you learn more complex queries in subsequent modules.

To create a table alias in a FROM clause, you will use syntax similar to several of the column alias techniques.

MCT USE ONLY STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

You may use the keyword AS to separate the table name from the alias. This style is preferred:

Table Alias Using AS

```
SELECT orderid, unitprice, qty
FROM Sales.OrderDetails AS OD;
```

You may omit the keyword AS and simply follow the table name with the alias:

Table Alias Without AS

```
SELECT orderid, unitprice, qty
FROM Sales.OrderDetails OD;
```

To combine table and column aliases in the same SELECT statement, use the following approach:

Table and Column Aliases Combined

```
SELECT OD.orderid, OD.unitprice, OD.qty AS Quantity
FROM Sales.OrderDetails AS OD;
```

Note: There is no table alias equivalent to the use of the equals sign (=) in a column alias.

Since this module focuses on single-table queries, you might not yet see a benefit to using table aliases. In the next module, you will learn how to retrieve data from multiple tables in a single SELECT statement. In those queries, the use of table aliases to represent table names will become quite useful.

The Impact of Logical Processing Order on Aliases

An issue may arise in the use of column aliases. Aliases created in the SELECT clause may not be referred to in others in the query, such as a WHERE or HAVING clause. This is due to the logical order query processing. The WHERE and HAVING clauses are processed before the SELECT clause and its aliases are evaluated. An exception to this is the ORDER BY clause.

- FROM, WHERE, and HAVING clauses processed before SELECT
- Aliases created in SELECT clause only visible to ORDER BY
- Expressions aliased in SELECT clause may be repeated elsewhere in query

An example is provided here for illustration and will run without error:

ORDER BY With Alias

```
SELECT orderid, unitprice, qty AS quantity
FROM Sales.OrderDetails
ORDER BY quantity;
```

MCT USE ONLY. STUDENT USE PROHIBITED

However, the following example will return an error, as the WHERE clause has been processed before the SELECT clause defines the alias:

Incorrect WHERE With Alias

```
SELECT orderid, unitprice, qty AS quantity
FROM Sales.OrderDetails
WHERE quantity > 10;
```

The resulting error message:

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

As a result, you will often need to repeat an expression more than once—in the SELECT clause, where you may create an alias to name the column, and in the WHERE or HAVING clause:

Correct WHERE With Alias

```
SELECT orderid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE YEAR(orderdate) = '2008'
```

Additionally, within the SELECT clause, you may not refer to a column alias that was defined in the same SELECT statement, regardless of column order.

The following statement will return an error:

Column Alias used in SELECT Clause

```
SELECT productid, unitprice AS price, price * qty AS total
FROM Sales.OrderDetails;
```

The resulting error:

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

Demonstration: Using Column and Table Aliases

In this demonstration, you will see how to:

- Use column and table aliases.

Demonstration Steps

Use Column and Table Aliases

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod03\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod03\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

Question: You have the following query:

```
SELECT FirstName LastName FROM HumanResources.Employees;
```

You are surprised to find that the query returns the following

```
LastName
---------
Fred
Rosalind
Anil
Linda
```

What error have you made in the SELECT query?

Lesson 4
Writing Simple CASE Expressions

A CASE expression extends the ability of a SELECT clause to manipulate data as it is retrieved. Often when writing a query, you need to substitute a value from a column of a table with another value. While you will learn how to perform this kind of lookup from another table later in this course, you can also perform basic substitutions using simple CASE expressions in the SELECT clause. In real-world environments, CASE is often used to help make cryptic data held in a column more meaningful.

A CASE expression returns a scalar (single-valued) value based on conditional logic, often with multiple conditions. As a scalar value, it may be used wherever single values can be used. Besides the SELECT statement, CASE expressions can be used in WHERE, HAVING, and ORDER BY clauses.

Lesson Objectives

In this lesson you will learn how to:

- Understand the use of CASE expressions in SELECT clauses.
- Understand the simple form of a CASE expression.

Using CASE Expressions in SELECT Clauses

In T-SQL, CASE expressions return a single, or scalar, value. Unlike some other programming languages, in T-SQL CASE, expressions are not statements, nor do they specify the control of programmatic flow. Instead, they are used in SELECT clauses (and other clauses) to return the result of an expression. The results appear as a calculated column and should be aliased for clarity.

In T-SQL queries, CASE expressions are often used to provide an alternative value for one stored in the source table. For example, a CASE expression might be used to provide a friendly text name for something stored as a compact numeric code.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Forms of CASE Expressions

In T-SQL, CASE expressions may take one of two forms – simple CASE or searched (Boolean) CASE.

Simple CASE expressions, the subject of this lesson, compare an input value to a list of possible matching values:

1. If a match is found, the first matching value is returned as the result of the CASE expression. Multiple matches are not permitted.
2. If no match is found, a CASE expression returns the value found in an ELSE clause, if one exists.
3. If no match is found and no ELSE clause is present, the CASE expression returns a NULL.

- Two forms of T-SQL CASE expressions:
- Simple CASE
  - Compares one value to a list of possible values
  - Returns first match
  - If no match, returns value found in optional ELSE clause
  - If no match and no ELSE, returns NULL
- Searched CASE
  - Evaluates a set of predicates, or logical expressions
  - Returns value found in THEN clause matching first expression that evaluates to TRUE

For example, the following CASE expression substitutes a descriptive category name for the categoryid value stored in the Production.Categories table. Note that this is not a JOIN operation, just a simple substitution using a single table:

CASE Expression

```
SELECT productid, productname, categoryid,
CASE categoryid
      WHEN 1 THEN 'Beverages'
      WHEN 2 THEN 'Condiments'
      WHEN 2 THEN 'Confections'
      ELSE 'Unknown Category'
END AS categoryname
FROM Production.Categories
```

The results:

```
productid productname  categoryid  categoryname
--------- ------------ ---------- --------------------
101       Tea          1          Beverages
102       Mustard      2          Condiments
103       Dinner Rolls 9          Unknown Category
```

Note: The preceding example is presented for illustration only and will not run against the sample databases provided with the course.

Searched (Boolean) CASE expressions compare an input value to a set of logical predicates or expressions. The expression can contain a range of values to match against. Like a simple CASE expression, the return value is found in the THEN clause of the matching value.

Due to their dependence on predicate expressions, which will not be covered until later in this course, further discussion of searched CASE expressions is beyond the scope of this lesson. See *CASE (Transact-SQL)* in Books Online:

CASE (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402717

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Using a Simple CASE Expression

In this demonstration, you will see how to:

- Use a simple CASE expression.

Demonstration Steps

Use a Simple CASE Expression

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod03\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod03\Demo folder.
3. In Solution Explorer, open the **41 – Demonstration D.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

   Question: You have the following SELECT query:

   ```
   SELECT FirstName, LastName, Sex FROM HumanResources.Employees;
   ```

   This returns:

   ```
   FirstName  LastName  Sex
   ---------- --------- ----
   Maya       Steele    1
   Adam       Brookes   0
   Naomi      Sharp     1
   Pedro      Fielder   0
   Zachary    Parsons   0
   ```

   How could you make these results clearer?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Writing Basic SELECT Statements

Scenario

You are a business analyst for Adventure Works who will be writing reports using corporate databases stored in SQL Server 2014. You have been provided with a set of business requirements for data and will write basic T-SQL queries to retrieve the specified data from the databases.

Objectives

After completing this lab, you will be able to:

- Write simple SELECT statements.
- Eliminate duplicate rows by using the DISTINCT keyword.
- Use table and column aliases.
- Use a simple CASE expression.

Estimated Time: 40 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Writing Simple SELECT Statements

Scenario

As a business analyst, you want a better understanding of your corporate data. Usually the best approach for an initial project is to get an overview of the main tables and columns involved so you can better understand different business requirements. After an initial overview, you will have to provide a report for the marketing department because staff there would like to send invitation letters for a new campaign. You will use the TSQL sample database.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. View all the Tables in the TSQL Database in Object Explorer
3. Write a Simple SELECT Statement
4. Write a SELECT Statement that Includes Specific Columns

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab03\Starter folder as Administrator.

▶ Task 2: View all the Tables in the TSQL Database in Object Explorer

1. Using SSMS, connect to MIA-SQL using Windows authentication (if you are connecting to an on-premises instance of SQL Server) or SQL Server authentication.
2. In Object Explorer, expand the TSQL database and expand the Tables folder.
3. Take a look at the names of the tables in the Sales schema.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Write a Simple SELECT Statement

1. Open the project file D:\Labfiles\Lab03\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement that will return all rows and all columns from the Sales.Customers table.

Note: You can use drag-and-drop functionality to drag items like table and column names from Object Explorer to the query window. Write the same SELECT statement using the drag-and-drop functionality.

3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\52 - Lab Exercise 1 - Task 2 Result.txt.

▶ Task 4: Write a SELECT Statement that Includes Specific Columns

1. Expand the Sales.Customers table in Object Explorer and expand the Columns folder. Observe all columns in the table.
2. Write a SELECT statement to return the contactname, address, postalcode, city, and country columns from the Sales.Customers table.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\53 - Lab Exercise 1 - Task 3 Result.txt.
4. What is the number of rows affected by the last query? (Tip: Because you are issuing a SELECT statement against the whole table, the number of rows will be the same as that for the whole Sales.Customers table).

Results: After this exercise, you should know how to create simple SELECT statements to analyze existing tables.

Exercise 2: Eliminating Duplicates Using DISTINCT

Scenario

After supplying the marketing department with a list of all customers for a new campaign, you are asked to provide a list of all the different countries that the customers come from.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement that Includes a Specific Column
2. Write a SELECT Statement that Uses the DISTINCT Clause

▶ Task 1: Write a SELECT Statement that Includes a Specific Column

1. Open the project file D:\Labfiles\Lab03\Starter\Project\Project.ssmssln and T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement against the Sales.Customers table showing only the country column.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Write a SELECT Statement that Uses the DISTINCT Clause

1. Copy the SELECT statement in Task 1 and modify it to return only distinct values.
2. Execute the written statement and compare the results that you achieved with the desired results shown in file D:\Labfiles\Lab03\Solution\63 - Lab Exercise 2 - Task 2 Result.txt.
3. How many rows did the query in Task 1 return?
4. How many rows did the query in Task 2 return?
5. Under which circumstances do the following queries against the Sales.Customers table return the same result?

```
SELECT city, region FROM Sales.Customers;
SELECT DISTINCT city, region FROM Sales.Customers;
```

6. Is the DISTINCT clause being applied to all columns specified in the query or just the first column?

Results: After this exercise, you should have an understanding of how to return only the different (distinct) rows in the result set of a query.

Exercise 3: Using Table and Column Aliases

Scenario

After getting the initial list of customers, the marketing department would like to have more readable titles for the columns and a list of all products in the TSQL database.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement that Uses a Table Alias
2. Write A SELECT Statement That Uses Column Aliases
3. Write a SELECT Statement that Uses a Table Alias and a Column Alias
4. Analyze and Correct the Query

▶ Task 1: Write a SELECT Statement that Uses a Table Alias

1. Open the project file D:\Labfiles\Lab03\Starter\Project\Project.ssmssln and T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to return the contactname and contacttitle columns from the Sales.Customers table, assigning "C" as the table alias. Use the table alias C to prefix the names of the two needed columns in the SELECT list. The benefit of using table aliases will become clearer in future modules when topics such as joins and subqueries will be introduced.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab03\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write A SELECT Statement That Uses Column Aliases

1. Write a SELECT statement to return the contactname, contacttitle, and companyname columns. Assign these with the aliases Name, Title, and Company Name, respectively, in order to return more human-friendly column titles for reporting purposes.

2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\73 - Lab Exercise 3 - Task 2 Result.txt. Notice specifically the titles of the columns in the desired output.

▶ Task 3: Write a SELECT Statement that Uses a Table Alias and a Column Alias

1. Write a query to display the productname column from the Production.Products table using "P" as the table alias and Product Name as the column alias.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\74 - Lab Exercise 3 - Task 3 Result.txt.

▶ Task 4: Analyze and Correct the Query

1. A developer has written a query to retrieve two columns (city and region) from the Sales.Customers table. When the query is executed, it returns only one column. Your task is to analyze the query, correct it to return two columns, and explain why the query returned only one.

```
SELECT city country
FROM Sales.Customers;
```

2. Execute the query exactly as written inside a query window and observe the result.
3. Correct the query to return the city and country columns from the Sales.Customers table.

Why did the query return only one column? What was the title of the column in the output? What is the best practice when using aliases for columns to avoid such errors?

Results: After this exercise, you will know how to use aliases for table and column names.

Exercise 4: Using a Simple CASE Expression

Scenario

Your company has a long list of products, and the members of the marketing department would like to have product category information in their reports. They have supplied you with a document containing the following mapping between the product category IDs and their names:

categoryid	Categoryname
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

They have an active marketing campaign, and would like to include product category information in their reports.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement

2. Write a SELECT Statement that Uses a CASE Expression

3. Write a SELECT Statement that Uses a CASE Expression to differentiate Campaign-Focused Products

▶ Task 1: Write a SELECT Statement

1. Open the project file D:\Labfiles\Lab03\Starter\Project\Project.ssmssln and T-SQL script 81 Lab Exercise 4.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to display the categoryid and productname columns from the Production.Products table.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement that Uses a CASE Expression

1. Enhance the SELECT statement in task 1 by adding a CASE expression that generates a result column named categoryname. The new column should hold the translation of the category ID to its respective category name, based on the mapping table supplied earlier. Use the value "Other" for any category IDs not found in the mapping table.
2. Execute the written statement and compare the results that you achieved with the desired output shown in the file D:\Labfiles\Lab03\Solution\83 - Lab Exercise 4 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement that Uses a CASE Expression to differentiate Campaign-Focused Products

1. Modify the SELECT statement in task 2 by adding a new column named iscampaign. This will show the description "Campaign Products" for the categories Beverages, Produce, and Seafood and the description "Non-Campaign Products" for all other categories.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\84 - Lab Exercise 4 - Task 3 Result.txt.

Results: After this exercise, you should know how to use CASE expressions to write simple conditional logic.

Module Review and Takeaways

Best Practice: Terminate all T-SQL statements with a semicolon. This will make your code more readable, avoid certain parsing errors, and protect your code against changes in future versions of SQL Server.
Consider standardizing your code on the AS keyword for labeling column and table aliases. This will make it easier to read and avoids accidental aliases.

Review Question(s)

Question: Why is the use of SELECT * not a recommended practice?

Question: What will happen if you omit a comma between column names in a SELECT clause?

Question: What kind of result does a simple CASE statement return?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 4

Querying Multiple Tables

Contents:

Module Overview	**4-1**
Lesson 1: Understanding Joins	**4-2**
Lesson 2: Querying with Inner Joins	**4-7**
Lesson 3: Querying with Outer Joins	**4-11**
Lesson 4: Querying with Cross Joins and Self Joins	**4-15**
Lab: Querying Multiple Tables	**4-19**
Module Review and Takeaways	**4-24**

Module Overview

In real-world environments, it is likely that the data you need to query is stored in multiple locations. Earlier, you learned how to write basic single-table queries. In this module, you will learn how to write queries that combine data from multiple sources in Microsoft SQL Server. You will do so by writing queries containing joins, which allow you to retrieve data from two (or more) tables based on data relationships between the tables.

Objectives

After completing this module, you will be able to:

- Describe how multiple tables may be queried in a SELECT statement using joins.
- Write queries that use inner joins.
- Write queries that use outer joins.
- Write queries that use self joins and cross joins.

Lesson 1
Understanding Joins

In this lesson, you will learn the fundamentals of joins in SQL Server. You will discover how the FROM clause in a T-SQL SELECT statement creates intermediate virtual tables that will be consumed by subsequent phases of the query. You will learn how an unrestricted combination of rows from two tables yields a Cartesian product. This module also covers the common join types in T-SQL multi-table queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the relationship between the FROM Clause and virtual tables in a SELECT statement.
- Describe a Cartesian product and how it may be created by a join.
- Describe the common join types in T-SQL queries.
- Understand the difference between ANSI SQL-89 and SQL-92 join syntax.

The FROM Clause and Virtual Tables

Earlier, you learned about the logical order of operations performed when SQL Server processes a query. You will recall that the FROM clause of a SELECT statement is the first phase to be processed. This clause determines which table or tables will be the source of rows for the query. As you will see in this module, this will hold true whether you are querying a single table or bringing together multiple tables as the source of your query.

- FROM clause determines source tables to be used in SELECT statement
- FROM clause can contain tables and operators
- Result set of FROM clause is virtual table
  - Subsequent logical operations in SELECT statement consume this virtual table
- FROM clause can establish table aliases for use by subsequent phases of query

To learn about the additional capabilities of the FROM clause, it will be useful to think of the clause function as creating and populating a virtual table. This virtual table will hold the output of the FROM clause and be used subsequently by other phases of the SELECT statement, such as the WHERE clause. As you add extra functionality, such as join operators, to a FROM clause, it will be helpful to think of the purpose of the FROM clause elements as either to add rows to, or remove rows from, the virtual table.

Note: The virtual table created by a FROM clause is a logical entity only. In SQL Server, no physical table is created, whether persistent or temporary, to hold the results of the FROM clause, as it is passed to the WHERE clause or other subsequent phases.

The syntax for the SELECT statement you have used for earlier queries in this course has appeared as follows:

SELECT Syntax

```
SELECT ...
FROM <table> AS <alias>;
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

You learned earlier that the FROM clause is processed first, and as a result, any table aliases you create there may be referenced in the SELECT clause. You will see numerous examples of table aliases in this module. While they are optional, except in the case of self join queries, you will quickly see how they can be a convenient tool when writing queries. Compare the following two queries, which have the same output, but which differ in their use of aliases. (Note that the examples use a JOIN clause, which will be covered later in this module).

The first query uses no table aliases:

Without Table Aliases

```
USE TSQL ;
GO
SELECT Sales.Orders.orderid, Sales.Orders.orderdate,
            Sales.OrderDetails.productid,Sales.OrderDetails.unitprice,
            Sales.OrderDetails.qty
FROM  Sales.Orders
           JOIN Sales.OrderDetails ON Sales.Orders.orderid = Sales.OrderDetails.orderid ;
```

The second example retrieves the same data but uses table aliases:

With Table Aliases

```
USE TSQL ;
GO
SELECT o.orderid, o.orderdate,
            od.productid, od.unitprice,
            od.qty
FROM  Sales.Orders AS o
JOIN    Sales.OrderDetails AS od ON o.orderid = od.orderid ;
```

As you can see, the use of table aliases improves the readability of the query, without affecting the performance. It is strongly recommended that you use table aliases in your multi-table queries.

Note: Once a table has been designated with an alias in the FROM clause, it is a best practice to use the alias when referring to columns from that table in other clauses.

Join Terminology: Cartesian Product

When learning about writing multi-table queries in T-SQL, it is important to understand the concept of Cartesian products. In mathematics, this is the product of two sets. The product of a set of two items and a set of six is a set of 12 items – or 6 x 2. In databases, a Cartesian product is the result of joining every row of one input table to all rows of another input table. The product of a table with 10 rows and a table with 100 rows is a result set with 1,000 rows.

For most T-SQL queries, a Cartesian product is not the desired outcome. Typically, a Cartesian product occurs when two input tables are joined without considering any logical relationships between them. In the absence of any information about relationships, the SQL Server query processor will output all possible combinations of rows.

While this can have some practical applications, such as creating a table of numbers or generating test data, it is not typically useful and can have severe performance effects. You will learn a useful application of Cartesian joins later in this module.

Note: In the next topic, you will compare two different methods for specifying the syntax of a join. You will see that one method may lead you toward writing accidental Cartesian product queries.

Overview of Join Types

To populate the virtual table produced by the FROM clause in a SELECT statement, SQL Server uses join operators. These add or remove rows from the virtual table, before it is handed off to subsequent logical phases of the SELECT statement:

- Join types in FROM clauses specify the operations performed on the virtual table:

Join Type	Description
Cross	Combines all rows in both tables (creates Cartesian product)
Inner	Starts with Cartesian product; applies filter to match rows between tables based on predicate
Outer	Starts with Cartesian product; all rows from designated table preserved, matching rows from other table retrieved. Additional NULLs inserted as placeholders

- A cross join operator (CROSS JOIN) adds all possible combinations of the two input tables' rows to the virtual table. Any filtering of the rows will happen in a WHERE clause. For most querying purposes, this operator is to be avoided.
- An inner join operator (INNER JOIN, or just JOIN) first creates a Cartesian product, and then filters the results using the predicate supplied in the ON clause, removing any rows from the virtual table that do not satisfy the predicate. The inner join is a very common type of join for retrieving rows with attributes that match across tables, such as matching Customers to Orders by a common custid.
- An outer join operator (LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN) first creates a Cartesian product, and like an inner join, filters the results to find rows that match in each table. However, all rows from one table are preserved, and added back to the virtual table after the initial filter is applied. NULLs are placed on attributes where no matching values are found.

Note: Unless otherwise qualified with CROSS or OUTER, the JOIN operator defaults to an INNER join.

MCT USE ONLY. STUDENT USE PROHIBITED

T-SQL Syntax Choices

Through the history of versions of SQL Server, the product has changed to keep pace with variations in the ANSI standards for the SQL language. One of the most notable places where these changes are visible is in the syntax for the join operator in a FROM clause.

In ANSI SQL-89, no ON operator was defined. Joins were represented in a comma-separated list of tables, and any filtering, such as for an inner join, was performed in the WHERE clause. This syntax is still supported by SQL Server, but due to the complexity of representing the filters for an outer join in the WHERE clause, as well as any other filtering, it is not recommended to use this. Additionally, if a WHERE clause is accidentally omitted, ANSI SQL-89-style joins can easily become Cartesian products and cause performance problems.

The following queries illustrate this syntax and this potential problem:

Cartesian Product

```
USE TSQL;
GO
/*
This is ANSI SQL-89 syntax for an inner join, with the filtering performed in the WHERE
clause.
*/
SELECT c.companyname, o.orderdate
FROM Sales.Customers AS c, Sales.Orders AS o
WHERE c.custid = o.custid;
....
(830 row(s) affected)

/*
This is ANSI SQL-89 syntax for an inner join, omitting the WHERE clause and causing an
inadvertent Cartesian join.
*/

SELECT c.companyname, o.orderdate
FROM Sales.Customers AS c, Sales.Orders AS o;
...
(75530 row(s) affected)
```

With the advent of the ANSI SQL-92 standard, support for the ON clause was added. T-SQL also supports this syntax. Joins are represented in the FROM clause by using the appropriate JOIN operator. The logical relationship between the tables, which becomes a filter predicate, is represented with the ON clause.

The following example restates the previous query with the newer syntax:

JOIN Clause

```
SELECT c.companyname, o.orderdate
FROM Sales.Customers AS c JOIN Sales.Orders AS o
ON c.custid = o.custid;
```

Note: The ANSI SQL-92 syntax makes it more difficult to create accidental Cartesian joins. Once the keyword JOIN has been added, a syntax error will be raised if an ON clause is missing.

MCT USE ONLY STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Understanding Joins

In this demonstration, you will see how to:

- Use Joins.

Demonstration Steps

Use Joins

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod04\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod04\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.

Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Question	

You have the following T-SQL query:
 	
`SELECT o.ID AS OrderID, o.CustomerName, p.ProductName, p.ModelNumber,`
`FROM Sales.Orders AS o`
`JOIN Sales.Products AS p`
`ON o.ProductID = p.ID;`
 	
Which of the following types of JOIN will the query perform?	

Select the correct answer.	
	A cross join
	An inner join
	An outer left join
	An outer right join

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Querying with Inner Joins

In this lesson, you will learn how to write inner join queries, the most common type of multi-table query in a business environment. By expressing a logical relationship between the tables, you will retrieve only those rows with matching attributes present in both.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe inner joins.
- Write queries using inner joins.
- Describe the syntax of an inner join.

Understanding Inner Joins

T-SQL queries that use inner joins are the most common types to solve many business problems, especially in highly normalized database environments. To retrieve data that has been stored across multiple tables, you will often need to reassemble it via inner join queries.

- Returns only rows where a match is found in both input tables
- Matches rows based on attributes supplied in predicate
  - ON clause in SQL-92 syntax (preferred)
  - WHERE clause in SQL-89 syntax
- Why filter in ON clause?
  - Logical separation between filtering for purposes of join and filtering results in WHERE
  - Typically no difference to query optimizer
- If join predicate operator is =, also known as equi-join

As you have previously learned, an inner join begins its logical processing phase as a Cartesian product, which is then filtered to remove any rows that don't match the predicate.

In SQL-89 syntax, that predicate is in the WHERE clause. In SQL-92 syntax, that predicate is within the FROM clause in the ON clause:

SQL-89 and SQL-92 Join Syntax Compared

```
--ANSI SQL-89 syntax
SELECT c.companyname, o.orderdate
FROM Sales.Customers AS c, Sales.Orders AS o
WHERE c.custid = o.custid;

--ANSI SQL-92 syntax
SELECT c.companyname, o.orderdate
FROM Sales.Customers AS c JOIN Sales.Orders AS o
ON c.custid = o.custid;
```

From a performance standpoint, you will find that the query optimizer in SQL Server does not favor one syntax over the other. However, as you learn about additional types of joins, especially outer joins, you will likely decide that you prefer to use the SQL-92 syntax and filter in the ON clause. Keeping the join filter logic in the ON clause and leaving other data filtering in the WHERE clause, will make your queries easier to read and test.

Using the ANSI SQL-92 syntax, let's examine the steps by which SQL Server will logically process this query. Line numbers are added for clarity and are not submitted to the query engine for execution:

ANSI-92 Join

```
1)    SELECT c.companyname, o.orderdate
2)    FROM Sales.Customers AS c
3)    JOIN Sales.Orders AS o
4)    ON c.custid = o.custid;
```

As you learned earlier, the FROM clause will be processed before the SELECT clause. Therefore, let's track the processing, beginning with line 2:

- The FROM clause designates the Sales.Customers table as one of the input tables, giving it the alias of 'c'.
- The JOIN operator in line 3 reflects the use of an INNER join (the default type in T-SQL) and designates Sales.Orders as the other input table, which has an alias of 'o'.
- SQL Server will perform a logical Cartesian join on these tables and pass the results to the next phase in the virtual table. (Note that the physical processing of the query may not actually perform the Cartesian product operation, depending on the optimizer's decisions).
- Using the ON clause, SQL Server will filter the virtual table, retaining only those rows where a custid value from the 'c' table (Sales.Customers has been replaced by the alias) matches a custid from the 'p' table (Sales.Orders has been replaced by an alias).
- The remaining rows are left in the virtual table and handed off to the next phase in the SELECT statement. In this example, the virtual table is next processed by the SELECT clause, and only two columns are returned to the client application.
- The result? A list of customers who have placed orders. Any customers who have never placed an order have been filtered out by the ON clause, as have any orders that happen to have a customer ID that doesn't correspond to an entry in the customer list.

Inner Join Syntax

When writing queries using inner joins, consider the following guidelines:

- As you have seen, table aliases are preferred not only for the SELECT list, but also for expressing the ON clause.
- Inner joins may be performed on a single matching attribute, such as an orderid, or on multiple matching attributes, such as the combination of orderid and productid. Joins that match multiple attributes are called composite joins.

- The order in which tables are listed and joined in the FROM clause does not matter to the SQL Server optimizer. (This will not be the case for OUTER JOIN queries in the next topic). Conceptually, joins will be evaluated from left to right.
- Use the JOIN keyword once for each two tables in the FROM list. For a two-table query, specify one join. For a three-table query, you will use JOIN twice – once between the first two tables, and once again between the output of the first two and the third table.

MCT USE ONLY. STUDENT USE PROHIBITED

Inner Join Examples

The following are some examples of inner joins:

This query performs a join on a single matching attribute, relating the categoryid from the Production.Categories table to the categoryid from the Production.Products table:

Inner Join Example

```
SELECT c.categoryid, c.categoryname,
p.productid, p.productname
FROM Production.Categories AS c
JOIN Production.Products AS p
ON c.categoryid = p.categoryid;
```

This query performs a composite join on two matching attributes, relating city and country attributes from Sales.Customers to HR.Employees. Note the use of the DISTINCT operator to filter out duplicate occurrences of city, country:

Inner Join Example

```
SELECT DISTINCT e.city, e.country
FROM Sales.Customers AS c
JOIN HR.Employees AS e
ON c.city = e.city AND c.country = e.country;
```

Note: The demonstration code for this lesson also uses the DISTINCT operator to filter duplicates.

This next example shows how an inner join may be extended to include more than two tables. Note that the Sales.OrderDetails table is joined not to the Sales.Orders table, but to the output of the JOIN between Sales.Customers and Sales.Orders. Each instance of JOIN...ON performs its own population and filtering of the virtual output table. It is up to the SQL Server query optimizer to decide in which order the joins and filtering will be performed.

This next example shows how an inner join may be extended to include more than two tables.

Inner Join Example

```
SELECT c.custid, c.companyname, o.orderid, o.orderdate, od.productid, od.qty
FROM Sales.Customers AS c
JOIN Sales.Orders AS o
ON c.custid = o.custid
JOIN Sales.OrderDetails AS od
ON o.orderid = od.orderid;
```

Demonstration: Querying with Inner Joins

In this demonstration, you will see how to:

- Use inner joins.

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration Steps

Use Inner Joins

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod04\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod04\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

   Question: You have the following T-SQL query:

```
SELECT HumanResources.Employees.ID, HumanResources.Employers.ID AS
CompanyID,
   HumanResources.Employees.Name, HumanResources.Employers.Name AS
CompanyName
FROM HumanResources.Employees
JOIN HumanResources.Employers
ON HumanResources.Employees.EmployerID =
HumanResources.Employers.ID;
```

   How can you improve the readability of this query?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Querying with Outer Joins

In this lesson, you will learn how to write queries that use outer joins. While not as common as inner joins, the use of outer joins in a multi-table query can provide an alternative view of your business data. As with inner joins, you will express a logical relationship between the tables. However, you will retrieve not only rows with matching attributes, but all rows present in one of the tables, whether or not there is a match in the other table.

Lesson Objectives

After completing this lesson, you will be able to:

- Understand the purpose and function of outer joins.
- Be able to write queries using outer joins.
- Be able to combine an OUTER JOIN operator in a FROM clause with a nullability test in a WHERE clause to reveal non-matching rows.

Understanding Outer Joins

In the previous lesson, you learned how to use inner joins to match rows in separate tables. As you saw, SQL Server built the results of an inner join query by filtering out rows that failed to meet the conditions expressed in the ON clause predicate. The result is that only rows that matched from both tables were displayed. With an outer join, you may choose to display all the rows from one table, along with those that match from the second table. Let's look at an example, then explore the process.

- Returns all rows from one table and any matching rows from second table
- One table's rows are "preserved"
  - Designated with LEFT, RIGHT, FULL keyword
  - All rows from preserved table output to result set
- Matches from other table retrieved
- Additional rows added to results for non-matched rows
  - NULLs added in places where attributes do not match
- Example: Return all customers and for those who have placed orders, return order information. Customers without matching orders will display NULL for order details.

First, let's examine the following query, written as an inner join:

Inner Join

```
USE AdventureWorks;
GO
SELECT c.CustomerID, soh.SalesOrderID
FROM Sales.Customer AS c JOIN Sales.SalesOrderHeader AS soh
ON c.CustomerID = soh.CustomerID
--(31465 row(s) affected)
```

Note that this example uses the AdventureWorks2008R2 database for these samples. When written as an inner join, the query returns 31,465 rows. These rows represent a match between customers and orders. Only those CustomerIDs that are in both tables will appear in the results. Only customers who have placed orders will be returned.

Now, let's examine the following query, written as an outer left join:

Outer Left Join

```
USE AdventureWorks;
GO

SELECT c.CustomerID, soh.SalesOrderID
FROM Sales.Customer AS c LEFT OUTER JOIN Sales.SalesOrderHeader AS soh
ON c.CustomerID = soh.CustomerID
--(32166 row(s) affected)
```

This example uses a LEFT OUTER JOIN operator, which as you will learn, directs the query processor to preserve all rows from the table on the left (Sales.Customer) and displays the SalesOrderID values for matching rows in Sales.SalesOrderHeader. However, there are more rows returned in this example. All customers are returned, whether or not they have placed an order. As you will see in this lesson, an outer join will display all the rows from one side of the join or another, whether or not they match.

What does an outer join query display in columns where there was no match? In this example, there are no matching orders for 701 customers. In the place of the SalesOrderID column, SQL Server will output NULL where values are otherwise missing.

Outer Join Syntax

When writing queries using outer joins, consider the following guidelines:

- As you have seen, table aliases are preferred not only for the SELECT list, but also for expressing the ON clause.
- Outer joins are expressed using the keywords LEFT, RIGHT, or FULL preceding OUTER JOIN. The purpose of the keyword is to indicate which table (on which side of the keyword JOIN) should be preserved and have all its rows displayed, match or no match.
- As with inner joins, outer joins may be performed on a single matching attribute, such as an orderid, or on multiple matching attributes, such as orderid and productid.
- Unlike inner joins, the order in which tables are listed and joined in the FROM clause does matter, as it will determine whether you choose LEFT or RIGHT for your join.
- Multi-table joins are more complex when an OUTER JOIN is present. The presence of NULLs in the results of an outer join may cause issues if the intermediate results are then joined, via an inner join, to a third table. Rows with NULLs may be filtered out by the second join's predicate.
- To display only rows where no match exists, add a test for NULL in a WHERE clause following an OUTER JOIN predicate.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Outer Join Examples

The following are some examples of outer joins:

This query displays all customers and provides information about each of their orders if any exist:

Outer Join Example

```
USE TSQL;
GO
SELECT c.custid, c.companyname, o.orderid,
o.orderdate
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o
ON c.custid =o.custid;
```

- All customers with order details if present:

```
SELECT c.custid, c.contactname, o.orderid,
o.orderdate
FROM Sales.Customers AS C
LEFT OUTER JOIN Sales.Orders AS O
    ON c.custid = o.custid;
```

- Customers that did not place orders:

```
SELECT c.custid, c.contactname, o.orderid,
o.orderdate
FROM Sales.Customers AS C LEFT OUTER JOIN
Sales.Orders AS O
    ON c.custid = o.custid
WHERE o.orderid IS NULL;
```

This query displays only customers who have never placed an order:

Outer Join Example

```
SELECT c.custid, c.companyname, o.orderid, o.orderdate
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o
ON c.custid =o.custid
WHERE o.orderid IS NULL;
```

Demonstration: Querying with Outer Joins

In this demonstration, you will see how to:

- Use outer joins.

Demonstration Steps

Use Outer Joins

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod04\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod04\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Question
You have a table named PoolCars and a table named Bookings in your ResourcesScheduling database. You want to return all the pool cars for which there are zero bookings. Which of the following queries should you use?

Question	
Select the correct answer.	
	SELECT pc.ID, pc.Make, pc.Model, pc.LicensePlate FROM ResourcesScheduling.PoolCars AS pc, ResourcesScheduling.Bookings AS b WHERE pc.ID = b.CarID;
	SELECT pc.ID, pc.Make, pc.Model, pc.LicensePlate FROM ResourcesScheduling.PoolCars AS pc RIGHT OUTER JOIN ResourcesScheduling.Bookings AS b ON pc.ID = b.CarID;
	SELECT pc.ID, pc.Make, pc.Model, pc.LicensePlate FROM ResourcesScheduling.PoolCars AS pc JOIN ResourcesScheduling.Bookings AS b ON pc.ID = b.CarID;
	SELECT pc.ID, pc.Make, pc.Model, pc.LicensePlate FROM ResourcesScheduling.PoolCars AS pc LEFT OUTER JOIN ResourcesScheduling.Bookings AS b ON pc.ID = b.CarID WHERE b.BookingID IS NULL;

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 4
Querying with Cross Joins and Self Joins

In this lesson, you will learn about additional types of joins, which are useful in some more specialized scenarios.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe a use for a cross join.
- Write queries that use cross joins.
- Describe a use for a self join.
- Write queries that use self joins.

Understanding Cross Joins

Cross join queries create a Cartesian product that, as you have learned in this module so far, are to be avoided. Although you have seen a means to create one with ANSI SQL-89 syntax, you haven't seen how or why to do so with ANSI SQL-92. This topic will revisit cross joins and Cartesian products.

- Combine each row from first table with each row from second table
- All possible combinations output
- Logical foundation for inner and outer joins
  - Inner join starts with Cartesian product, adds filter
  - Outer join takes Cartesian output, filtered, adds back non-matching rows (with NULL placeholders)
- Due to Cartesian product output, not typically a desired form of join
- Some useful exceptions:
  - Table of numbers, generating data for testing

To explicitly create a Cartesian product, you would use the CROSS JOIN operator.

This will create a result set with all possible combinations of input rows:

Cross Join

```
SELECT ...
FROM table1 AS t1 CROSS JOIN table2 AS t2;
```

While this is not typically a desired output, there are a few practical applications for writing an explicit cross join:

- Creating a table of numbers, with a row for each possible value in a range.
- Generating large volumes of data for testing. When cross joined to itself, a table with as few as 100 rows can readily generate 10,000 output rows with very little work on your part.

Cross Join Syntax

When writing queries with CROSS JOIN, consider the following:

- There is no matching of rows performed, and therefore no ON clause is required.
- To use ANSI SQL-92 syntax, separate the input table names with the CROSS JOIN operator.

- No matching performed, no ON clause used
- Return all rows from left table combined with each row from right table (ANSI SQL-92 syntax):

```
SELECT ...
FROM t1 CROSS JOIN t2
```

- Return all rows from left table combined with each row from right table (ANSI SQL-89 syntax):

```
SELECT ...
FROM t1, t2
```

Cross Join Examples

The following is an example of using CROSS JOIN to create all combinations of two input sets:

Using the TSQL sample, this will take nine employee first and last names to generate 81 combinations:

Cross Join Example

```
SELECT e1.firstname, e2.lastname
FROM HR.Employees e1 CROSS JOIN
HR.Employees e2;
```

- Create test data by returning all combinations of two inputs:

```
SELECT e1.firstname, e2.lastname
FROM HR.Employees AS e1
CROSS JOIN HR.Employees AS e2;
```

Understanding Self Joins

To this point, the joins you have learned about have involved separate multiple tables. There may be scenarios in which you need to compare and retrieve data stored in the same table. For example, in a classic human resources application, an Employees table might include information about the supervisor of each employee in the employee's own row. Each supervisor is also listed as an employee. To retrieve the employee information and match it to the related supervisor, you can use the table twice in your query, joining it to itself for the purposes of the query.

- Why use self joins?
  - Compare rows in same table to each other
- Create two instances of same table in FROM clause
  - At least one alias required
- Example: Return all employees and the name of the employee's manager

There are other scenarios in which you will want to compare rows within a table with one another. As you have seen, it's fairly easy to compare columns within the same row using T-SQL, but how to compare values from different rows (such as a row which stores a starting time with another row in the same table that stores a corresponding stop time) is less obvious. Self joins are a useful technique for these types of queries.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

In order to accomplish tasks like this, you will want to consider the following guidelines:

- Create two instances of the same table in the FROM clause, and join them as needed, using inner or outer joins.
- Use table aliases to create two separate aliases for the same table. At least one of these must have an alias.
- Use the ON clause to provide a filter using separate columns from the same table.

The following example, which you will examine closely in the next topic, illustrates these guidelines:

This query retrieves employees and their matching manager information from the Employees table joined to itself:

Self Join Example

```
SELECT e.empid ,e.lastname AS empname,e.title,e.mgrid, m.lastname AS mgrname
  FROM HR.Employees AS e
  JOIN HR.Employees AS m
  ON e.mgrid=m.empid;
```

This yields results like the following:

```
empid empname      title                mgrid mgrname
----- ------------ -------------------- ----- -------
2     Funk         Vice President, Sales 1     Davis
3     Lew          Sales Manager         2     Funk
4     Peled        Sales Representative  3     Lew
5     Buck         Sales Manager         2     Funk
6     Suurs        Sales Representative  5     Buck
7     King         Sales Representative  5     Buck
8     Cameron      Sales Representative  3     Lew
9     Dolgopyatova Sales Representative  5     Buck
```

Self Join Examples

The following are some examples of self joins:

This query returns all employees, along with the name of each employee's manager, when a manager exists (inner join). Note that an employee with no manager listed will be missing from the results:

Self Join Example

```
SELECT e.empid ,e.lastname AS
empname,e.title,e.mgrid, m.lastname AS
mgrname
  FROM HR.Employees AS e
  JOIN HR.Employees AS m
  ON e.mgrid=m.empid;
```

- Return all employees with ID of employee's manager when a manager exists (inner join):

```
SELECT e.empid, e.lastname,
       e.title, e.mgrid, m.lastname
FROM   HR.Employees AS e
JOIN HR.Employees AS m
ON e.mgrid = m.empid;
```

- Return all employees with ID of manager (outer join). This will return NULL for the CEO:

```
SELECT e. empid, e.lastname,
       e.title, m.mgrid
FROM HR.Employees AS e
LEFT OUTER JOIN HR.Employees AS m
ON e.mgrid = m.empid;
```

This query returns all employees with the name of each manager (outer join). This restores the missing employee, who turns out to be a CEO with no manager:

Self Join Example

```
SELECT e.empid ,e.lastname AS empname,e.title,e.mgrid, m.lastname AS mgrname
  FROM HR.Employees AS e
  LEFT OUTER JOIN HR.Employees AS m
  ON e.mgrid=m.empid;
```

Demonstration: Querying with Cross Joins and Self Joins

In this demonstration, you will see how to:

- Use self joins and cross joins.

Demonstration Steps

Use Self Joins and Cross Joins

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod04\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod04\Demo folder.
3. In Solution Explorer, open the **41 – Demonstration D.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

   Question: You have two tables named FirstNames and LastNames. You want to generate a set of fictitious full names from this data. There are 150 entries in the FirstNames table and 250 entries in the LastNames table. You use the following query:

   SELECT (f.Name + ' ' + l.Name) AS FullName

   FROM FirstNames AS f

   CROSS JOIN LastNames AS l

   How many fictitious full names will be returned by this query?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Querying Multiple Tables

Scenario

You are a business analyst for Adventure Works, who will be writing reports using corporate databases stored in SQL Server. You have been provided with a set of business requirements for data and you will write T-SQL queries to retrieve the specified data from the databases. You notice that the data is stored in separate tables, so you will need to write queries using various join operations.

Objectives

After completing this lab, you will be able to:

- Write queries that use inner joins.
- Write queries that use multiple-table inner joins.
- Write queries that use self joins.
- Write queries that use outer joins
- Write queries that use cross joins.

Estimated Time: 50 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Writing Queries That Use Inner Joins

Scenario

You no longer need the mapping information between categoryid and categoryname that was supplied because you now have the Production.Categories table with the needed mapping rows. Write a SELECT statement using an inner join to retrieve the productname column from the Production.Products table and the categoryname column from the Production.Categories table.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment

2. Write a SELECT Statement that Uses an Inner Join

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab04\Starter folder as Administrator.

▶ Task 2: Write a SELECT Statement that Uses an Inner Join

1. Open the project file D:\Labfiles\Lab04\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement that will return the productname column from the Production.Products table (use table alias 'p') and the categoryname column from the Production.Categories table (use table alias 'c') using an inner join.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab04\Solution\52 - Lab Exercise 1 - Task 2 Result.txt.

4. Which column did you specify as a predicate in the ON clause of the join? Why?
5. Let us say that there is a new row in the Production.Categories table and this new product category does not have any products associated with it in the Production.Products table. Would this row be included in the result of the SELECT statement written in task 1? Please explain.

Results: After this exercise, you should know how to use an inner join between two tables.

Exercise 2: Writing Queries That Use Multiple-Table Inner Joins

Scenario

The sales department would like a report of all customers that placed at least one order, with detailed information about each one. A developer prepared an initial SELECT statement that retrieves the custid and contactname columns from the Sales.Customers table and the orderid column from the Sales.Orders table. You should observe the supplied statement and add additional information from the Sales.OrderDetails table.

The main tasks for this exercise are as follows:

1. Execute the T-SQL Statement
2. Apply the Needed Changes and Execute the T-SQL Statement
3. Change the Table Aliases
4. Add an Additional Table and Columns

▶ Task 1: Execute the T-SQL Statement

1. Open the project file D:\Labfiles\Lab04\Starter\Project\Project.ssmssln and the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. The developer has written this query:

```
SELECT
custid, contactname, orderid
FROM Sales.Customers
INNER join Sales.Orders ON Customers.custid = Orders.custid;
```

3. Execute the query exactly as written inside a query window and observe the result.
4. You get an error. What is the error message? Why do you think this happened?

▶ Task 2: Apply the Needed Changes and Execute the T-SQL Statement

1. Notice that there are full source table names written as table aliases.
2. Apply the needed changes to the SELECT statement so that it will run without an error. Test the changes by executing the T-SQL statement.
3. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\62 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Change the Table Aliases

1. Copy the T-SQL statement from task 2 and modify it to use the table aliases 'C' for the Sales.Customers table and 'O' for the Sales.Orders table.
2. Execute the written statement and compare the results with those in task 2.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

3. Change the prefix of the columns in the SELECT statement with full source table names and execute the statement.
4. You get an error. Why?
5. Change the SELECT statement to use the table aliases written at the beginning of the task.

▶ Task 4: Add an Additional Table and Columns

1. Copy the T-SQL statement from task 3 and modify it to include three additional columns from the Sales.OrderDetails table: productid, qty, and unitprice.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\63 - Lab Exercise 2 - Task 4 Result.txt.

Results: After this exercise, you should have a better understanding of why aliases are important and how to do a multiple-table join.

Exercise 3: Writing Queries That Use Self Joins

Scenario

The HR department would like a report showing employees and their managers. They want to see the lastname, firstname, and title columns from the HR.Employees table for each employee and the same columns for the employee's manager.

The main tasks for this exercise are as follows:

1. Write a Basic SELECT Statement

2. Write a Query that Uses a Self Join

▶ Task 1: Write a Basic SELECT Statement

1. Open the project file D:\Labfiles\Lab04\Starter\Project\Project.ssmssln and the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. In order to better understand the needed tasks, you will first write a SELECT statement against the HR.Employees table showing the empid, lastname, firstname, title, and mgrid columns.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\72 - Lab Exercise 3 - Task 1 Result.txt. Notice the values in the mgrid column. The mgrid column is in a relationship with empid column. This is called a self-referencing relationship.

▶ Task 2: Write a Query that Uses a Self Join

1. Copy the SELECT statement from task 1 and modify it to include additional columns for the manager information (lastname, firstname) using a self join. Assign the aliases mgrlastname and mgrfirstname, respectively, to distinguish the manager names from the employee names.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\73 - Lab Exercise 3 - Task 2 Result.txt. Notice the number of rows returned.
3. Is it mandatory to use table aliases when writing a statement with a self join? Can you use a full source table name as an alias? Please explain.
4. Why did you get fewer rows in the T-SQL statement under task 2 compared to task 1?

MCT USE ONLY. STUDENT USE PROHIBITED

Results: After this exercise, you should have an understanding of how to write T-SQL statements that use self joins.

Exercise 4: Writing Queries That Use Outer Joins

Scenario

The sales department was satisfied with the report you produced in exercise 2. Now sales staff would like to change the report to show all customers, even if they did not have any orders, and still include order information for the customers who did. You need to write a SELECT statement to retrieve all rows from Sales.Customers (columns custid and contactname) and the orderid column from the table Sales.Orders.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement that Uses an Outer Join

▶ Task 1: Write a SELECT Statement that Uses an Outer Join

1. Open the project file D:\Labfiles\Lab04\Starter\Project\Project.ssmssln and the T-SQL script 81 - Lab Exercise 4.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the custid and contactname columns from the Sales.Customers table and the orderid column from the Sales.Orders table. The statement should retrieve all rows from the Sales.Customers table.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.
4. Notice the values in the column orderid. Are there any missing values (marked as NULL)? Why?

Results: After this exercise, you should have a basic understanding of how to write T-SQL statements that use outer joins.

Exercise 5: Writing Queries That Use Cross Joins

Scenario

The HR department would like to prepare a personalized calendar for each employee. The IT department supplied you with T-SQL code that will generate a table with all dates for the current year. Your job is to write a SELECT statement that would return all rows in this new calendar date table for each row in the HR.Employees table.

The main tasks for this exercise are as follows:

1. Execute the T-SQL Statement
2. Write a SELECT Statement that Uses a Cross Join
3. Drop the HR.Calendar Table

▶ Task 1: Execute the T-SQL Statement

1. Open the project file D:\Labfiles\Lab04\Starter\Project\Project.ssmssln and the T-SQL script 91 - Lab Exercise 5.sql. Ensure that you are connected to the TSQL database.
2. Execute the T-SQL code under task 1. Don't worry if you do not understand the provided T-SQL code, as it is used here to provide a more realistic example for a cross join in the next task.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Write a SELECT Statement that Uses a Cross Join

3. Write a SELECT statement to retrieve the empid, firstname, and lastname columns from the HR.Employees table and the calendardate column from the HR.Calendar table.
4. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\92 - Lab Exercise 5 - Task 2 Result.txt.

   Note: The dates from the query might not exactly match the solution file.
5. What is the number of rows returned by the query? There are nine rows in the HR.Employees table. Try to calculate the total number of rows in the HR.Calendar table.

▶ Task 3: Drop the HR.Calendar Table

1. Execute the provided T-SQL statement to remove the HR.Calendar table.

Results: After this exercise, you should have an understanding of how to write T-SQL statements that use cross joins.

Module Review and Takeaways

Best Practice: Table aliases should always be defined when joining tables. Joins should be expressed using SQL-92 syntax, with JOIN and ON keywords.

Review Question(s)

Question: How does an inner join differ from an outer join?

Question: Which join types include a logical Cartesian product?

Question: Can a table be joined to itself?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 5

Sorting and Filtering Data

Contents:

Module Overview	**5-1**
Lesson 1: Sorting Data	**5-2**
Lesson 2: Filtering Data with Predicates	**5-6**
Lesson 3: Filtering Data with TOP and OFFSET-FETCH	**5-10**
Lesson 4: Working with Unknown Values	**5-16**
Lab: Sorting and Filtering Data	**5-19**
Module Review and Takeaways	**5-24**

Module Overview

In this module, you will learn how to enhance queries to limit the rows they return and control the order in which the rows are displayed.

Earlier in this course, you learned that, according to relational theory, sets of data do not include any definition of a sort order. As a result, if you require the output of a query to be displayed in a certain order, you will need to add an ORDER BY clause to your SELECT statement. In this module, you will learn how to write queries using ORDER BY to control the display order.

In a previous module, you also learned how to build a FROM clause to return rows from one or more tables. It is unlikely that you will always want to return all rows from the source. For performance reasons, as well as the needs of your client application or report, you will want to limit which rows are returned. As you will learn in this module, you can limit rows with a WHERE clause based on a predicate, or you can limit rows with TOP and OFFSET-FETCH, based on the number of rows and ordering.

As you work with real-world data in your queries, you may encounter situations where values are missing. It is important to write queries that can handle missing values correctly. In this module, you will learn about handling missing and unknown results.

Lesson 1
Sorting Data

In this lesson, you will learn how to add an ORDER BY clause to your queries to control the order of rows displayed in the query's output.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the ORDER BY clause.
- Describe the ORDER BY clause syntax.
- List examples of the ORDER BY clause.

Using the ORDER BY Clause

In the logical order of query processing, ORDER BY is the last phase of a SELECT statement to execute. ORDER BY provides the ability to control the sorting of rows as they are output from the query to the client application. Without an ORDER BY clause, Microsoft® SQL Server® does not guarantee the order of rows, in keeping with relational theory.

- ORDER BY sorts rows in results for presentation purposes
  - No guaranteed order of rows without ORDER BY
  - Use of ORDER BY guarantees the sort order of the result
  - Last clause to be logically processed
  - Sorts all NULLs together
- ORDER BY can refer to:
  - Columns by name, alias or ordinal position (not recommended)
  - Columns not part of SELECT list
  - Unless DISTINCT specified
- Declare sort order with ASC or DESC

To sort the output of your query, you will add an ORDER BY clause in this form:

ORDER BY Clause

```
SELECT <select_list>
FROM   <table_source>
ORDER BY <order_by_list> ASC|DESC;
```

ORDER BY can take several types of elements in its list:

- Columns by name. Additional columns beyond the first specified in the list will be used as tiebreakers for non-unique values in the first column.
- Column aliases. Remember that ORDER BY is processed after the SELECT clause and therefore has access to aliases defined in the SELECT list.
- Columns by position in the SELECT clause. This is not recommended, due to diminished readability and the extra care required to keep the ORDER BY list up to date with any changes made to the SELECT list column order.
- Columns not detailed in the SELECT list, but part of tables listed in the FROM clause. If the query uses a DISTINCT option, any columns in the ORDER BY list must be found in the SELECT list.

Note: ORDER BY may also include a COLLATE clause, which provides a way to sort by a specific character collation, and not the collation of the column in the table. Collations will be further discussed later in this course.

In addition to specifying which columns should be used to determine the sort order, you may also control the direction of the sort through the use of ASC for ascending (A-Z, 0-9) or DESC for descending (Z-A, 9-

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

0). Ascending sorts are the default. Each column may be provided with a separate order, as in the following example.

Employees will be listed from most recent hire to least recent, with employees hired on the same date listed alphabetically by last name:

Ascending and Descending Sort

```
USE TSQL;
GO
SELECT hiredate, firstname, lastname
FROM HR.Employees
ORDER BY hiredate DESC, lastname ASC;
```

Additional documentation on the ORDER BY clause can be found in Books Online at:

ORDER BY Clause (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402718

ORDER BY Clause Syntax

The syntax of the ORDER BY clause appears as follows:

ORDER BY Clause

```
ORDER BY <order_by_list>
OFFSET <offset_value> ROW|ROWS FETCH
FIRST|NEXT <fetch_value> ROW|ROWS ONLY
```

Note: The use of the OFFSET-FETCH option in the ORDER BY clause will be covered later in this module.

- Writing ORDER BY using column names:

```
SELECT <select list>
FROM <table source>
ORDER BY <column1_name>, <column2_name>;
```

- Writing ORDER BY using column aliases:

```
SELECT <column> AS <alias>
FROM <table source>
ORDER BY <alias>;
```

- Specifying sort order in the ORDER BY clause:

```
SELECT <column> AS <alias>
FROM <table source>
ORDER BY <column_name|alias> ASC|DESC;
```

Most variations of ORDER BY will occur in the ORDER BY list. To specify columns by name, with the default ascending order, use the following syntax:

ORDER BY List

```
ORDER BY <column_name_1>, <column_name_2>;
```

A fragment of code using columns from the Sales.Customers table would look like this:

ORDER BY List Example

```
ORDER BY country, region, city;
```

To specify columns by aliases defined in the SELECT clause, use the following syntax:

ORDER BY an Alias

```
SELECT <column_name_1> AS alias1, <column_name_2> AS alias2
FROM <table source>
ORDER BY alias1;
```

A query for the Sales.Orders table using column aliases would look like this:

ORDER BY With Alias Example

```
SELECT orderid, custid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
ORDER BY orderyear;
```

Note: See the previous topic for the syntax and usage of ASC or DESC to control sort order.

ORDER BY Clause Examples

The following are examples of common queries using ORDER BY to sort the output for display. All queries use the TSQL sample database.

- ORDER BY with column names:

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate;
```

- ORDER BY with column alias:

```
SELECT orderid, custid, YEAR(orderdate) AS
orderyear
FROM Sales.Orders
ORDER BY orderyear;
```

- ORDER BY with descending order:

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

A query against the Sales.Orders table, sorting the results by the orderdate column, specified by name:

ORDER BY Example

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate;
```

A query against the Sales.Orders table, which defines an alias in the SELECT clause and sorts by that column's alias:

ORDER BY Example

```
SELECT orderid, custid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
ORDER BY orderyear DESC;
```

A query against the Sales.Orders table, which sorts the output in descending order of orderdate (that is, most recent to oldest):

ORDER BY Example

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

A query against the HR.Employees table, which sorts the employees in descending order of hire date (that is, most recent to oldest), using lastname to differentiate employees hired on the same date:

ORDER BY Example

```
SELECT hiredate, firstname, lastname
FROM HR.Employees
ORDER BY hiredate DESC, lastname ASC;
```

MCT USE ONLY STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Sorting Data

In this demonstration, you will see how to:

- Sort data using the ORDER BY clause,

Demonstration Steps

Sort Data Using the ORDER BY Clause

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod05\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod05\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

   Question: Describe why, if you declare an alias for a column in the SELECT clause, you cannot use that alias in the WHERE clause, but you can use it in the ORDER BY clause.

Lesson 2
Filtering Data with Predicates

When querying SQL Server, you will mostly want to retrieve only a subset of all the rows stored in the table listed in the FROM clause. This is especially true as data volumes grow. To limit which rows are returned, you will typically use the WHERE clause in the SELECT statement. In this lesson, you will learn how to construct WHERE clauses to filter out rows that do not match the predicate.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the WHERE clause.
- Describe the syntax of the WHERE clause.

Filtering Data in the WHERE Clause with Predicates

In order to limit the rows that are returned by your query, you will need to add a WHERE clause to your SELECT statement, following the FROM clause. WHERE clauses are constructed from a search condition, which in turn is written as a predicate expression. The predicate provides a logical filter through which each row must pass. Only rows returning TRUE in the predicate will be output to the next logical phase of the query.

- WHERE clauses use predicates
  - Must be expressed as logical conditions
  - Only rows for which predicate evaluates to TRUE are accepted
  - Values of FALSE or UNKNOWN filtered out
- WHERE clause follows FROM, precedes other clauses
  - Can't see aliases declared in SELECT clause
- Can be optimized by SQL Server to use indexes
- Data filtered server-side
  - Can reduce network traffic and client memory usage

When writing a WHERE clause, keep the following considerations in mind:

- Your predicate must be expressed as a logical condition, evaluating to TRUE or FALSE. (This will change when working with missing values or NULL. See Lesson 4 for more information.)
- Only rows for which the predicate evaluates as TRUE will be passed through the filter.
- Values of FALSE or UNKNOWN will be filtered out.
- Column aliases declared in the query's SELECT clause cannot be used in the WHERE clause predicate.

Remember that, logically, the WHERE clause is the next phase in query execution after FROM, so will be processed before other clauses such as SELECT. One consequence of this is that the WHERE clause will be unable to refer to column aliases created in the SELECT clause. If you have created expressions in the SELECT list, you will need to repeat them to use in the WHERE clause.

For example, the following query, which uses a simple calculated expression in the SELECT list, will execute properly:

Filtering Example

```
SELECT orderid, custid, YEAR(orderdate) AS ordyear
FROM Sales.Orders
WHERE YEAR(orderdate) = 2006;
```

MCT USE ONLY. STUDENT USE PROHIBITED

The following query will fail, due to the use of column aliases in the WHERE clause:

Incorrect Column Alias in WHERE Clause

```
SELECT orderid, custid, YEAR(orderdate) AS ordyear
FROM Sales.Orders
WHERE ordyear = 2006;
```

The error message points to the use of the column alias in Line 3 of the batch:

```
Msg 207, Level 16, State 1, Line 3
Invalid column name 'ordyear'.
```

From the perspective of query performance, the use of effective WHERE clauses can provide a significant impact on SQL Server. Rather than return all rows to the client for post-processing, a WHERE clause causes SQL Server to filter data on the server side. This can reduce network traffic and memory usage on the client. SQL Server developers and administrators can also create indexes to support commonly-used predicates, furthering improving performance.

WHERE Clause Syntax

In Books Online, the syntax of the WHERE clause appears as follows:

WHERE Clause Syntax

```
WHERE <search_condition>
```

The most common form of a WHERE clause is as follows:

Typical WHERE Clause

```
WHERE <column> <operator> <expression>
```

- Filter rows for customers from Spain

```
SELECT contactname, country
FROM Sales.Customers
WHERE country = N'Spain';
```

- Filter rows for orders after July 1, 2007

```
SELECT orderid, orderdate
FROM Sales.Orders
WHERE orderdate > '20070101';
```

- Filter orders within a range of dates

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20070101' AND orderdate < '20080101';
```

For example, the following code fragment shows a WHERE clause that will filter only customers from Spain:

WHERE Clause Example

```
SELECT contactname, country
FROM Sales.Customers
WHERE country = N'Spain';
```

Any of the logical operators introduced in the T-SQL language module earlier in this course may be used in a WHERE clause predicate.

This example filters orders placed after a specified date:

WHERE CLAUSE Example

```
SELECT orderid, orderdate
FROM Sales.Orders
WHERE orderdate > '20070101';
```

Note: The representation of dates as strings delimited by quotation marks will be covered in the next module.

MCT USE ONLY. STUDENT USE PROHIBITED

In addition to using logical operators, literals, or constants in a WHERE clause, you may also use several T-SQL options in your predicate:

Predicates and Operators	Description
IN	Determines whether a specified value matches any value in a subquery or a list.
BETWEEN	Specifies an inclusive range to test.
LIKE	Determines whether a specific character string matches a specified pattern.
AND	Combines two Boolean expressions and returns TRUE only when both are TRUE.
OR	Combines two Boolean expressions and returns TRUE if either is TRUE.
NOT	Reverses the result of a search condition.

Note: The use of LIKE to match patterns in character-based data will be covered in the next module.

The following example shows the use of the OR operator to combine conditions in a WHERE clause:

WHERE With OR Example

```
SELECT custid, companyname, country
FROM Sales.Customers
WHERE country = N'UK' OR country = N'Spain';
```

The following example modifies the previous query to use the IN operator for the same results:

WHERE with IN Example

```
SELECT custid, companyname, country
FROM Sales.Customers
WHERE country IN (N'UK',N'Spain');
```

The following example uses the NOT operator to reverse the previous condition:

NOT Operator

```
SELECT custid, companyname, country
FROM Sales.Customers
WHERE country NOT IN (N'UK',N'Spain');
```

The following example uses logical operators to search within a range of dates:

Range Example

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20070101' AND orderdate <= '20080101';
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The following example accomplishes the same results using the BETWEEN operator:

BETWEEN Operator

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
WHERE orderdate BETWEEN '20061231' AND '20080101';
```

Note: The use of comparison operators with date and time data types requires special consideration. For more information, see the next module.

Demonstration: Filtering Data with Predicates

In this demonstration, you will see how to:

- Filter data in a WHERE clause.

Demonstration Steps

Filter Data in a WHERE Clause

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod05\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod05\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

   Question: You have a table named Employees that includes a column named StartDate. You want to find who started in any year other than 2014. What query would you use?

Lesson 3
Filtering Data with TOP and OFFSET-FETCH

In the previous lesson, you wrote queries that filtered rows based on data stored within them. You can also write queries that filter ranges of rows, based either on a specific number to retrieve or one range of rows at a time. In this lesson, you will learn how to limit ranges of rows in the SELECT clause using a TOP option. You will also learn how to limit ranges of rows using the OFFSET-FETCH option of an ORDER BY clause.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the TOP option.
- Describe the OFFSET-FETCH clause.
- Describe the syntax of the OFFSET-FETCH clause.

Filtering in the SELECT Clause Using the TOP Option

When returning rows from a query, you may need to limit the total number of rows returned, as well as filter with a WHERE clause. The TOP option, a Microsoft-proprietary extension of the SELECT clause, will let you specify a number of rows to return, either as an ordinal number or percentage of all candidate rows.

The simplified syntax of the TOP option is as follows:

TOP Option

```
SELECT TOP (N) <column_list>
FROM  <table_source>
WHERE <search_condition>;
```

For example, to retrieve only the five most recent orders from the Sales.Orders table, use the following query:

TOP Example

```
SELECT TOP (5) orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Note that the TOP operator depends on an ORDER BY clause to provide meaningful precedence to the rows selected. In the absence of ORDER BY, there is no guarantee which rows will be returned. In the previous example, any five orders might be returned if there wasn't an ORDER BY clause.

In addition to specifying a fixed number of rows to be returned, the TOP keyword also accepts the WITH TIES option, which will retrieve any rows with values that might be found in the selected top N rows.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

For example, the following query will return five rows with the most recent order dates:

Without the WITH TIES Option

```
SELECT TOP (5) orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

The results show five rows with two distinct orderdate values:

```
orderid     custid      orderdate
----------- ----------- -----------------------
11077       65          2008-05-06 00:00:00.000
11076       9           2008-05-06 00:00:00.000
11075       68          2008-05-06 00:00:00.000
11074       73          2008-05-06 00:00:00.000
11073       58          2008-05-05 00:00:00.000
(5 row(s) affected)
```

However, by adding the WITH TIES option to the TOP clause, you will see that more rows qualify for the second-oldest order date:

With The WITH TIES Option

```
SELECT TOP (5) WITH TIES orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

This modified query returns the following results:

```
orderid     custid      orderdate
----------- ----------- -----------------------
11077       65          2008-05-06 00:00:00.000
11076       9           2008-05-06 00:00:00.000
11075       68          2008-05-06 00:00:00.000
11074       73          2008-05-06 00:00:00.000
11073       58          2008-05-05 00:00:00.000
11072       20          2008-05-05 00:00:00.000
11071       46          2008-05-05 00:00:00.000
11070       44          2008-05-05 00:00:00.000
 (8 row(s) affected)
```

The decision to include WITH TIES will depend on your knowledge of the source data and its potential for unique values.

To return a percentage of the row count, use the PERCENT option with TOP instead of a fixed number.

For example, if the Sales.Orders table contains 830 orders, the following query will return 83 rows:

Returning a Percentage of Records

```
SELECT TOP (10) PERCENT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

TOP (N) PERCENT may also be used with the WITH TIES option.

Note: TOP (N) PERCENT will round up to the nearest integer for purposes of row counts.

Additional information about the TOP clause can be found in Books Online at:

TOP (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402719

Filtering in the ORDER BY Clause Using OFFSET-FETCH

While the TOP option is used by many SQL Server professionals as a method for retrieving only a certain range of rows, it also has its disadvantages:

- TOP is proprietary to T-SQL and SQL Server.
- TOP does not support skipping a range of rows.
- Since TOP depends on an ORDER BY clause, you cannot use one sort order to establish the rows filtered by TOP and another to determine the output display.

To address a number of these concerns, Microsoft added a new option to SQL Server 2012 – the OFFSET-FETCH extension to the ORDER BY clause.

OFFSET-FETCH is an extension to the ORDER BY clause:

- Allows filtering a requested range of rows
  - Dependent on ORDER BY clause
- Provides a mechanism for paging through results
- Specify number of rows to skip, number of rows to retrieve:

```
ORDER BY <order_by_list>
OFFSET <offset_value> ROW(S)
FETCH FIRST|NEXT <fetch_value> ROW(S) ONLY
```

- New option in SQL Server 2012
  - Based on draft SQL:2011 standard
  - Provides more compatibility than TOP

Like TOP, OFFSET-FETCH allows you to return only a range of the rows selected by your query. However, it adds the functionality to supply a starting point (an offset) and a value to specify how many rows you would like to return (a fetch value). This provides a convenient technique for paging through results.

When paging, you will need to consider that each query with an OFFSET-FETCH clause runs independently of any previous or subsequent query. There is no server-side state maintained, and you will need to track your position through a result set via client-side code.

As you will see in the next topic, OFFSET-FETCH has been written to allow a more natural English language syntax.

For more information about the OFFSET-FETCH clause, go to the section *Using OFFSET and FETCH to limit the rows returned* in Books Online at:

ORDER BY Clause (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402718

MCT USE ONLY STUDENT USE PROHIBITED

OFFSET-FETCH Syntax

The syntax for the OFFSET-FETCH clause is as follows:

- OFFSET value must be supplied
  - May be zero if no skipping is required
- The optional FETCH clause allows all rows following the OFFSET value to be returned
- Natural Language approach to code:
  - ROW and ROWS interchangeable
  - FIRST and NEXT interchangeable
- OFFSET value and FETCH value may be constants or expressions, including variables and parameters

```
OFFSET <offset_value> ROW|ROWS
FETCH FIRST|NEXT <fetch_value> ROW|ROWS [ONLY]
```

OFFSET FETCH Clause

```
OFFSET { integer_constant |
offset_row_count_expression } { ROW | ROWS
}
    [FETCH { FIRST | NEXT }
{integer_constant |
fetch_row_count_expression } { ROW | ROWS }
ONLY]
```

To use OFFSET-FETCH, you will supply a starting OFFSET value (which may be zero) and an optional number of rows to return, as in the following example:

This example will skip the first 10 rows and then return the next 10 rows, as determined by the order date:

OFFSET FETCH Example

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate, orderid DESC
OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;
```

As you can see in the syntax definition, the OFFSET clause is required, but the FETCH clause is not. If the FETCH clause is omitted, all rows following OFFSET will be returned. You will also find that the keywords ROW and ROWS are interchangeable, as are FIRST and NEXT, which allows a more natural syntax.

To ensure results accuracy, especially as you "move" from page to page of data, it is important to construct an ORDER BY clause that will provide unique ordering and yield a deterministic result. Although unlikely due to SQL Server's query optimizer, it is technically possible for a row to appear on more than one page unless the range of rows is deterministic.

Note: To use OFFSET-FETCH for paging, you may supply the OFFSET value as well as row count expressions in the form of variables or parameters. You will learn more about variables and stored procedure parameters in later modules of this course.

The following are some examples of using OFFSET-FETCH in T-SQL queries, all of which use the TSQL sample database:

To retrieve the 50 most recent rows as determined by the order date, this query starts with an offset of zero. It will return a result similar to a SELECT TOP(50) query:

OFFSET FETCH Example

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC
OFFSET 0 ROWS FETCH FIRST 50 ROWS ONLY;
```

MCT USE ONLY. STUDENT USE PROHIBITED

This query will retrieve rows 51-100 of a result set:

OFFSET FETCH Example

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC
OFFSET 50 ROWS FETCH NEXT 50 ROWS ONLY;
```

Note: Unlike those found in any previous modules, examples of OFFSET-FETCH must be executed by SQL Server 2012 or later. OFFSET-FETCH is not supported in SQL Server 2008 R2 or earlier.

Demonstration: Filtering Data with TOP and OFFSET-FETCH

In this demonstration, you will see how to:

- Filter data using TOP and OFFSET-FETCH.

Demonstration Steps

Filter Data Using TOP and OFFSET-FETCH

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod05\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod05\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Check Your Knowledge

Question	
You have a table named Products in your Sales database. You are creating a paged display of products in an application that shows 20 products on each page ordered by name. Which of the following queries would return the third page of products?	
Select the correct answer.	
	SELECT ProductID, ProductName, ProductNumber FROM Sales.Products ORDER BY ProductName ASC OFFSET 60 ROWS FETCH NEXT 20 ROWS ONLY
	SELECT ProductID, ProductName, ProductNumber FROM Sales.Products ORDER BY ProductName ASC OFFSET 40 ROWS FETCH NEXT 20 ROWS ONLY;
	SELECT TOP (20) ProductID, ProductName, ProductNumber FROM Sales.Products ORDER BY ProductName ASC
	SELECT TOP (20) WITH TIES ProductID, ProductName, ProductNumber FROM Sales.Products ORDER BY ProductName ASC

Lesson 4
Working with Unknown Values

Unlike traditional Boolean logic, predicate logic in SQL Server needs to account for missing values and deal with cases where the result of a predicate is unknown. In this lesson, you will learn how three-valued logic accounts for unknown and missing values, how SQL Server uses NULL to mark missing values, and how to test for NULL in your queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe three-valued logic.
- Describe the use of NULL in queries.

Three-Valued Logic

Earlier in this course, you learned that SQL Server uses predicate logic as a framework for logical tests that return TRUE or FALSE. This is true for logical expressions where all values being tested are present. If you know the values of both X and Y, you can safely determine whether X>Y is TRUE or FALSE.

- SQL Server uses NULLs to mark missing values
  - NULL can be "missing but applicable" or "missing but inapplicable"
    - Customer middle name: Not supplied, or doesn't have one?
- With no missing values, predicate outputs are TRUE or FALSE only (5 >2, 1=1)
- With missing values, outputs can be TRUE, FALSE or UNKNOWN (NULL > 99, NULL = NULL)
- Predicates return UNKNOWN when comparing missing value to another value, including another missing value

However, in SQL Server, not all data being compared may be present. You need to plan for and act on the possibility that some data is missing or unknown. Values in SQL Server may be missing but applicable, such as the value of a middle initial that has not been supplied for an employee. It may also be missing but inapplicable, such as the value of a middle initial for an employee who has no middle name. In both cases, SQL Server will mark the missing value as NULL. A NULL is neither TRUE nor FALSE but is a mark for UNKNOWN, which represents the third value in three-valued logic.

As discussed above, you can determine whether X>Y is TRUE or FALSE when you know the values of both X and Y. But what does SQL Server return for the expression X>Y when Y is missing? SQL Server will return an UNKNOWN, marked as NULL. You will need to account for the possible presence of NULL in your predicate logic, as well as in the values stored in columns marked with NULL. You will need to write queries that use three-valued logic to account for three possible outcomes – TRUE, FALSE, and UNKNOWN.

MCT USE ONLY. STUDENT USE PROHIBITED

Handling NULL in Queries

Once you have acquired a conceptual understanding of three-valued logic and NULL, you need to understand the different mechanisms SQL Server uses for handling NULLs. Keep in mind the following guidelines:

- Different components of SQL Server handle NULL differently
  - Query filters (ON, WHERE, HAVING) filter out UNKNOWNs
  - CHECK constraints accept UNKNOWNS
  - ORDER BY, DISTINCT treat NULLs as equals
- Testing for NULL
  - Use IS NULL or IS NOT NULL rather than = NULL or <> NULL

```
SELECT custid, city, region, country
FROM Sales.Customers
WHERE region IS NOT NULL;
```

- Query filters, such as ON, WHERE, and the HAVING clause, treat NULL like a FALSE result. A WHERE clause that tests for a <column_value> = N will not return rows when the comparison is FALSE. Nor will it return rows when either the column value or the value of N is NULL.

Note the output of the following queries:

ORDER BY Query that includes NULL in Results

```
SELECT empid, lastname, region
FROM HR.Employees
ORDER BY region ASC; --Ascending sort order explicitly included for clarity.
```

This returns the following, with all employees whose region is missing (marked as NULL) sorted first:

```
empid       lastname             region
----------- -------------------- ---------------
5           Buck                 NULL
6           Suurs                NULL
7           King                 NULL
9           Dolgopyatova         NULL
8           Cameron              WA
1           Davis                WA
2           Funk                 WA
3           Lew                  WA
4           Peled                WA
```

Note: A common question about controlling the display of NULL in queries is whether NULLs can be forced to the end of a result set. As you can see, the ORDER BY clause sorts the NULLs together and first, a behavior you cannot override.

- ORDER BY treats NULLs as if they were the same value and always sorts NULLs together, putting them first in a column. Make sure you test the results of any queries in which the column being used for sort order contains NULLs, and understand the impact of ascending and descending sorts on NULLs.
- In ANSI-compliant queries, a NULL is never equivalent to another value, even another NULL. Queries written to test NULL with an equality will fail to return correct results.

Note the following example:

Incorrectly Testing For NULL

```
SELECT empid, lastname, region
FROM HR.Employees
WHERE region = NULL;
```

MCT USE ONLY. STUDENT USE PROHIBITED

This returns inaccurate results:

```
empid       lastname             region
----------- -------------------- ---------------
 (0 row(s) affected)
```

- Use the IS NULL (or IS NOT NULL) operator rather than equals (not equals).

See the following example:

Correctly Testing For NULL

```
SELECT empid, lastname, region
FROM HR.Employees
WHERE region IS NULL;
```

This returns correct results:

```
empid       lastname             region
----------- -------------------- ---------------
5           Buck                 NULL
6           Suurs                NULL
7           King                 NULL
9           Dolgopyatova         NULL
```

(4 row(s) affected)

Demonstration: Working with NULL

In this demonstration, you will see how to:

- Test for NULL.

Demonstration Steps

Test for NULL

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod05\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod05\Demo folder.
3. In Solution Explorer, open the **41 – Demonstration D.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

   Question: You have the following query:

   ```
   SELECT e.Name, e.Age
   FROM HumanResources.Employees AS e
   WHERE YEAR(e.Age) < 1990;
   ```

   Several employees have asked for their age to be removed from the Human Resources database. Will the above query return these employees?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Sorting and Filtering Data

Scenario

You are an Adventure Works business analyst who will be writing reports using corporate databases stored in SQL Server. You have been provided with a set of data business requirements and will write T-SQL queries to retrieve the specified data from the databases. You will need to retrieve only some of the available data, and return it to your reports in a specified order.

Objectives

After completing this lab, you will be able to:

- Write queries that filter data using a WHERE clause.
- Write queries that sort data using an ORDER BY clause.
- Write queries that filter data using the TOP option.
- Write queries that filter data using an OFFSET-FETCH clause.

Estimated Time: 60 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Writing Queries That Filter Data Using a WHERE Clause

Scenario

The marketing department is working on several campaigns for existing customers and staff need to obtain different lists of customers, depending on several business rules. Based on these rules, you will write the SELECT statements to retrieve the needed rows from the Sales.Customers table.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement that Uses a WHERE Clause
3. Write a SELECT Statement that Uses an IN Predicate in the WHERE Clause
4. Write a SELECT Statement that Uses a LIKE Predicate in the WHERE Clause
5. Observe the T-SQL Statement Provided by the IT Department
6. Write a SELECT Statement to Retrieve those Customers Without Orders

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab05\Starter folder as Administrator.

▶ Task 2: Write a SELECT Statement that Uses a WHERE Clause

1. Open the project file D:\Labfiles\Lab05\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement that will return the custid, companyname, contactname, address, city, country, and phone columns from the Sales.Customers table. Filter the results to include only the customers from the country Brazil.

3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab05\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.

▶ Task 3: Write a SELECT Statement that Uses an IN Predicate in the WHERE Clause

1. Write a SELECT statement that will return the custid, companyname, contactname, address, city, country, and phone columns from the Sales.Customers table. Filter the results to include only customers from the countries Brazil, UK, and USA.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab05\Solution\53 - Lab Exercise 1 - Task 2 Result.txt.

▶ Task 4: Write a SELECT Statement that Uses a LIKE Predicate in the WHERE Clause

1. Write a SELECT statement that will return the custid, companyname, contactname, address, city, country, and phone columns from the Sales.Customers table. Filter the results to include only the customers with a contact name starting with the letter A.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab05\Solution\54 - Lab Exercise 1 - Task 3 Result.txt.

▶ Task 5: Observe the T-SQL Statement Provided by the IT Department

1. The IT department has written a T-SQL statement that retrieves the custid and companyname columns from the Sales.Customers table and the orderid column from the Sales.Orders table:

```
SELECT
c.custid, c.companyname, o.orderid
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON c.custid = o.custid AND c.city = 'Paris';
```

2. Execute the query and notice two things. First, the query retrieves all the rows from the Sales.Customers table. Second, there is a comparison operator in the ON clause, specifying that the city column should be equal to the value "Paris".
3. Copy the provided T-SQL statement and modify it to have a comparison operator for the city column in the WHERE clause. Execute the query.
4. Compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab05\Solution\55 - Lab Exercise 1 - Task 4 Result.txt.
5. Is the result the same as in the first T-SQL statement? Why? What is the difference between specifying the predicate in the ON clause and in the WHERE clause?

▶ Task 6: Write a SELECT Statement to Retrieve those Customers Without Orders

1. Write a T-SQL statement to retrieve customers from the Sales.Customers table that do not have matching orders in the Sales.Orders table. Matching customers with orders is based on a comparison between the customer's and the order's custid values. Retrieve the custid and companyname columns from the Sales.Customers table. (Hint: Use a T-SQL statement similar to the one in the previous task.)
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab05\Solution\56 - Lab Exercise 1 - Task 5 Result.txt.

Results: After this exercise, you should be able to filter rows of data from one or more tables by using WHERE predicates with logical operators.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Writing Queries That Sort Data Using an ORDER BY Clause

Scenario

The sales department would like a report showing all the orders with some customer information. An additional request is that the result be sorted by the order dates and the customer IDs. Remember from the previous modules that the order of the rows in the output of a query without an ORDER BY clause is not guaranteed. Because of this, you will have to write a SELECT statement that uses an ORDER BY clause.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement that Uses an ORDER BY Clause
2. Apply the Needed Changes and Execute the T-SQL Statement
3. Order the Result by the firstname Column

▶ Task 1: Write a SELECT Statement that Uses an ORDER BY Clause

1. Open the project file D:\Labfiles\Lab05\Starter\Project\Project.ssmssln and the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the custid and contactname columns from the Sales.Customers table and the orderid and orderdate columns from the Sales.Orders table. Filter the results to include only orders placed on or after April 1, 2008 (filter the orderdate column). Then sort the result by orderdate in descending order and custid in ascending order.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab05\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Apply the Needed Changes and Execute the T-SQL Statement

1. Someone took your T-SQL statement from lab 4 and added the following WHERE clause:

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid,
m.lastname AS mgrlastname, m.firstname AS mgrfirstname
FROM HR.Employees AS e
INNER JOIN HR.Employees AS m ON e.mgrid = m.empid
WHERE mgrlastname = 'Buck';
```

2. Execute the query exactly as written inside a query window and observe the result.
3. You get an error. What is the error message? Why do you think this happened? (Tip: Remember the logical processing order of the query).
4. Apply the needed changes to the SELECT statement so that it will run without an error. Test the changes by executing the T-SQL statement.
5. Observe and compare the results that you achieved with the recommended results shown in the D:\Labfiles\Lab05\Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Order the Result by the firstname Column

1. Copy the existing T-SQL statement from task 2 and modify it so that the result will return all employees and be ordered by the manager's first name. First, try to use the source column name, and then the alias column name.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab05\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.
3. Why were you able to use a source column or alias column name?

Results: After this exercise, you should know how to use an ORDER BY clause.

Exercise 3: Writing Queries That Filter Data Using the TOP Option

Scenario

The sales department would like to have some additional reports that show the last invoiced orders and the top 10 percent of the most expensive products being sold.

The main tasks for this exercise are as follows:

1. Writing Queries That Filter Data Using the TOP Clause
2. Use the OFFSET-FETCH Clause to Implement the Same Task
3. Write a SELECT Statement to Retrieve the Most Expensive Products

▶ Task 1: Writing Queries That Filter Data Using the TOP Clause

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement USE TSQL; and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT TOP (20)
orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Use the OFFSET-FETCH Clause to Implement the Same Task

1. In the query pane, type the following query after the task 2 description:

```
SELECT
orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC
OFFSET 0 ROWS FETCH FIRST 20 ROWS ONLY;
```

2. Remember that the OFFSET-FETCH clause was a new functionality in SQL Server 2012. Unlike the TOP clause, the OFFSET-FETCH clause must be used with the ORDER BY clause.
3. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve the Most Expensive Products

1. In the query pane, type the following query after the task 3 description:

```
SELECT TOP (10) PERCENT
   productname, unitprice
FROM Production.Products
ORDER BY unitprice DESC;
```

2. Implementing this task with the OFFSET-FETCH clause is possible but not easy because, unlike TOP, OFFSET-FETCH does not support a PERCENT option.
3. Highlight the written query and click **Execute**.

Results: After this exercise, you should have an understanding of how to apply the TOP option in the SELECT clause of a T-SQL statement.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 4: Writing Queries That Filter Data Using the OFFSET-FETCH Clause

Scenario

In this exercise, you will implement a paging solution for displaying rows from the Sales.Orders table because the total number of rows is high. In each page of a report, the user should only see 20 rows.

The main tasks for this exercise are as follows:

1. OFFSET-FETCH Clause to Fetch the First 20 Rows
2. Use the OFFSET-FETCH Clause to Skip the First 20 Rows
3. Write a Generic Form of the OFFSET-FETCH Clause for Paging

▶ Task 1: OFFSET-FETCH Clause to Fetch the First 20 Rows

1. Open the project file D:\Labfiles\Lab05\Starter\Project\Project.ssmssln and the T-SQL script 81 - Lab Exercise 4.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the custid, orderid, and orderdate columns from the Sales.Orders table. Order the rows by orderdate and ordered, and then retrieve the first 20 rows.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab05\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.

▶ Task 2: Use the OFFSET-FETCH Clause to Skip the First 20 Rows

1. Copy the SELECT statement in task 1 and modify the OFFSET-FETCH clause to skip the first 20 rows and fetch the next 20.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab05\Solution\83 - Lab Exercise 4 - Task 2 Result.txt.

▶ Task 3: Write a Generic Form of the OFFSET-FETCH Clause for Paging

1. You are given the parameters @pagenum for the requested page number and @pagesize for the requested page size. Can you figure out how to write a generic form of the OFFSET-FETCH clause using those parameters? (Do not worry about not being familiar with those parameters yet).

Module Review and Takeaways

Review Question(s)

Question: Does the physical order of rows in an SQL Server table guarantee any sort order in queries using the table?

Question: You have the following query:

```
SELECT p.PartNumber, p.ProductName, o.Quantity
FROM Sales.Products AS p
LEFT OUTER JOIN Sales.OrderItems AS o
ON p.ID = o.ProductID
ORDER BY o.Quantity ASC
```

You have one new product that has yet to receive any orders. Will this product appear at the top or the bottom of the results?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 6

Working with SQL Server 2014 Data Types

Contents:

Module Overview	**6-1**
Lesson 1: Introducing SQL Server 2014 Data Types	**6-2**
Lesson 2: Working with Character Data	**6-10**
Lesson 3: Working with Date and Time Data	**6-17**
Lab: Working with SQL Server 2014 Data Types	**6-23**
Module Review and Takeaways	**6-29**

Module Overview

To write effective queries in T-SQL, you will need to understand how Microsoft® SQL Server® stores different types of data. This is especially important if your queries not only retrieve data from tables but also perform comparisons, manipulate data, and implement other operations.

In this module, you will learn about the data types SQL Server uses to store data. In the first lesson, you will be introduced to many types of numeric and special-use data types. You will learn about conversions between data types and the importance of data type precedence. You will learn how to work with character-based data types, including functions that can be used to manipulate the data. You will also learn how to work with temporal data, or date and time data, including functions to retrieve and manipulate all or portions of a stored date.

Objectives

After completing this module, you will be able to:

- Describe numeric data types, type precedence, and type conversions.
- Write queries using character data types.
- Write queries using date and time data types.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Introducing SQL Server 2014 Data Types

In this lesson, you will explore many of the data types SQL Server uses to store data and learn how data types are converted between types.

Note: Character, date, and time data types are excluded from this lesson but will be covered later in the module.

If your focus in taking this course is to write queries for reports, you may wish to take note of which data types are used in your environment. You can then plan your reports and client applications with enough capacity to display the range of values held by the SQL Server data types. You may also need to plan for conversions in your queries to display SQL Server data in other environments.

If your focus is to continue into database development and administration, you may wish to take note of the similarities and differences within categories of data types, and plan your storage accordingly, as you create types and design parameters for stored procedures.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how SQL Server uses data types.
- Describe the attributes of numeric data types, as well as binary strings and other specialized data types.
- Describe data type precedence and its use in converting data between different data types.
- Describe the difference between implicit and explicit data type conversion.

SQL Server Data Types

SQL Server 2014 defines a set of system data types for storing data in columns, holding values temporarily in variables, operating on data in expressions, and passing as parameters in stored procedures. Data types specify the type, length, precision, and scale of data. Understanding the basic types of data in SQL Server is fundamental to writing queries in T-SQL, as well as designing tables and creating other objects in SQL Server.

- SQL Server associates columns, expressions, variables, and parameters with data types
- Data types determine what kind of data can be held:
  - Integers, characters, dates, money, binary strings, etc.
- SQL Server supplies built-in data types
- Developers can also define custom types
  - Aliases in T-SQL
  - User-defined types in .NET code

SQL Server Data Type Categories	
Exact numeric	Unicode characters
Approximate numeric	Binary strings
Date and time	Other
Character strings	

SQL Server supplies built-in data types of various categories. Developers may also extend the supplied set by creating aliases to built-in types and even by producing new user-defined types using the Microsoft .NET Framework. This lesson will focus on the built-in system data types.

MCT USE ONLY. STUDENT USE PROHIBITED

Other than character, date, and time types, which will be covered later in this module, SQL Server data types can be grouped into the following categories:

- **Exact numeric**. These data types store data with precision, either as:
  - Integers with varying degrees of capacity.
  - Decimals that allow you to specify the total number of digits stored and how many of those should be to the right of the decimal place.

  As you learn about these types, take note of the relationship between capacity and storage requirements.
- **Approximate numeric**. These data types allow inexact values to be stored, typically for use in scientific calculations.
- **Binary strings.** These data types allow binary data to be stored, such as bytestreams or hashes, to support custom applications.
- **Other data types.** This catch-all category includes special types such as uniqueidentifier and XML, which are sometimes used as column data types (and are therefore accessible to queries). It also includes data types that are not used for storage, but rather for special operations such as cursor manipulations or creating output tables for further processing. If you are a report writer, you may only encounter the uniqueidentifier and XML data types.

Numeric Data Types

- When working with exact numeric data, you will see that there are three basic subcategories of data types in SQL Server – exact numeric, decimal numeric, and approximate numeric. Each SQL Server data type falls into one of these categories.
- Exact numeric types include:

• Exact Numeric

Data type	Range	Storage (bytes)
tinyint	0 to 255	1
smallint	-32,768 to 32,768	2
int	2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647)	4
Bigint	-2^63 - 2^63-1 (+/- 9 quintillion)	8
bit	1, 0 or NULL	1
decimal/numeric	- 10^38 +1 through 10^38 – 1 when maximum precision is used	5-17
money	-922,337,203,685,477.5808 to 922,337,203,685,477.5807	8
smallmoney	- 214,748.3648 to 214,748.3647	4

- **Integers**, where the distinction between types relates to capacity and storage requirements. Note that the **tinyint** data type, for example, can only hold values between 0 and 255, for the storage cost of 1 byte. At the other end of the spectrum, the **bigint** data type can hold plus or minus 9 quintillion (a very large value) at the cost of 8 bytes. You will need to decide which integer data type offers the best fit for capacity versus storage. You will often see that the **int** data type has been selected because it provides the best tradeoff—a capacity of plus or minus 2 billion at the cost of 4 bytes.
- **Decimal and numeric**, which allow you to specify the total number of digits to be stored (precision) and the number of digits to the right of the decimal (scale). As with integers, the greater the range, the higher the storage cost. Note that, while decimal is ISO standards-compliant, decimal and numeric are equivalent to one another. Numeric is kept for compatibility with earlier versions of SQL Server.
- **Money and smallmoney**, which are designed to hold monetary values with a maximum of four places. You may find that your organization uses the decimal type instead of money for its greater flexibility and precision.

- **Bit**, which is a single-bit value used to store Boolean values or flags. Storage for a bit column is dependent on how many others there may be in a table, due to SQL Server optimizing their storage.

Go to the following topics in Books Online: *Decimal and Numeric (Transact SQL)* at:

decimal and numeric (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402721

Precision, Scale, and Length (Transact-SQL) at:

Precision, Scale, and Length (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402723

Data Types (Transact-SQL) at:

Data Types (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402724

SQL Server also supplies data types for approximate numeric values. The approximate numeric data types are less accurate but have more capacity than the exact numeric data types. The approximate numeric data types store values in scientific notation, which loses accuracy because of a lack of precision.

- **Float** takes an optional parameter of the number of digits to be stored after the decimal. This parameter is called the mantissa, the value of which determines the storage size of the float. If the mantissa is in the range 1 to 24, the float requires 4 bytes. If the mantissa is between 25 and 53, it requires 8 bytes.
- **Real** is an ISO synonym for float (24).

Go to the topic *Float and Real (Transact-SQL)* in Books Online at:

float and real (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402725

Binary String Data Types

Binary string data types allow a developer to store binary information, such as serialized files, images, bytestreams, and other specialized data. If you are considering using the binary data type, note the differences in range and storage compared with integers and character string data. You can choose between fixed-width and varying-width binary strings. The difference between these will be explained in the character data type lesson later in the module.

Data Type	Range	Storage (bytes)
binary(n)	1-8000 bytes	n bytes
varbinary(n)	1-8000 bytes	n bytes + 2
varbinary(MAX)	1-2.1 billion (approx) bytes	actual length + 2

The following example shows a number converted to a binary data type. (You will learn about the CAST function in the next module.) This query:

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Converting to Binary Data Type

```
SELECT CAST(12345 AS BINARY(4)) AS Result;
```

Returns the following:

```
Result
----------
0x00003039
```

Go to the topic *Binary and Varbinary (Transact-SQL)* in Books Online at:

binary and varbinary (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402726

Other Data Types

In addition to numeric and binary types, SQL Server also supplies some other data types, allowing you to store and process XML, generate globally unique identifiers (GUIDs), represent hierarchies, and more. Some of these have limited use, others are more generally useful:

Data Type	Range	Storage (bytes)	Remarks
rowversion	Auto-generated	8	Successor type to timestamp.
uniqueidentifier	Auto-generated	16	Globally unique identifier (GUID).
xml	0-2 GB	0-2 GB	Stores XML in native hierarchical structure.
cursor	N/A	N/A	Not a storage data type.
hierarchyid	N/A	Depends on content	Represents position in a hierarchy.
sql_variant	0-8000 bytes	Depends on content	Can store data of various data types.
table	N/A	N/A	Not a storage data type, used for query and programmatic operations.

- **Rowversion** is a binary value, auto-incrementing when a row in a table is inserted or updated. It does not actually store time data in a form that will be useful to you. Rowversion also has other limitations.
- **Uniqueidentifier** provides a mechanism for an automatically generated value that is unique across multiple systems. It is stored as a 16-byte value. Uniqueidentifier must be generated either by converting from a string (reducing the guarantee of uniqueness) or by using the NEWID() system function.

For example, this query:

Unique Identifier

```
SELECT NEWID() AS [GUID];
```

Returns:

```
GUID
------------------------------------
1C0E3B5C-EA7A-41DC-8E1C-D0A302B5E58B
```

- **XML** allows the storage and manipulation of eXtensible Markup Language data. This data type stores up to 2 GB of data per instance of the type.

Additional Reading: See course 20464D: *Developing Microsoft® SQL Server® Databases* for additional information on the XML data type.

- **Cursors** are listed here for completeness. A SQL Server cursor is not a data type for storing data, but rather for use in variables or stored procedures that reference a cursor object. Discussions of cursors are beyond the scope of this module.
- **Hierarchyid** is a data type used to store hierarchical position data, such as levels of an organizational chart or bill of materials. SQL Server stores hierarchy data as binary data and exposes it through built-in functions.

Additional Reading: Go to course 20464D: *Developing Microsoft® SQL Server® Databases* for additional information about the hierarchyid data type.

- SQL_variant is a column data type that can store other common data types. Its use is not a best practice for typical data storage and may indicate design problems. It is listed here for completeness.
- Table data types can be used to store the results of T-SQL statements for further processing later, such as in a subsequent statement in a query. You will learn more about table types later in this course. Note that table types cannot be used as a data type for a column (such as to store nested tables).

Information on all of SQL Server's data types can be found in Books Online starting at:

Data Types (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402724

Data Type Precedence

When combining or comparing different data types in your queries, such as in a WHERE clause, SQL Server will need to convert one value from its data type to that of the other value. Which data type is converted depends on the precedence between the two.

SQL Server defines a ranking of all its data types by precedence—between any two data types, one will have a lower precedence and the other a higher precedence. When converting, SQL Server will convert the lower data type to the higher one. Typically, this will happen implicitly, without the need for special code. However, it is important for you to have a basic understanding of this precedence arrangement so you know when you need to manually, or explicitly, convert data types to combine or convert them.

For example, here is a partial list of data types, ranked according to their precedence:

1. XML
2. Datetime2
3. Date
4. Time
5. Decimal

MCT USE ONLY. STUDENT USE PROHIBITED

6. Int
7. Tinyint
8. Nvarchar
9. Char

When combining or comparing two expressions with different data types, the one lower on this list will be converted to the type that is higher. In this example, the variable of type tinyint will be implicitly converted to int before being added to the int variable @myInt:

```
DECLARE @myTinyInt AS TINYINT = 25;
DECLARE @myInt as INT = 9999;
SELECT @myTinyInt + @myInt;
```

Note: Any implicit conversion is transparent to the user so, if it fails (such as when your operation requires converting from a higher to a lower precedence), you will need to explicitly convert the data type. You will learn how to use the CAST function for this purpose in the next module.

For more information and a complete list of types and their precedence, go to Books Online at:

Data Type Precedence (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402727

When Are Data Types Converted?

There are a number of scenarios in which data types may be converted when querying SQL Server:

- When data is moved, compared, or combined with other data.
- During variable assignment.
- When using any operator that involves two operands of different types.
- When T-SQL code explicitly converts one type to another, using a CAST or CONVERT function.

- Data type conversion scenarios
  - When data is moved, compared, or combined with other data
  - During variable assignment
- Implicit conversion
  - When comparing data of one type to another
  - Transparent to user

  WHERE <column of smallint type> = <value of int type>
- Explicit conversion
  - Uses CAST or CONVERT functions

  CAST(unitprice AS int)
- Not all conversions allowed by SQL Server

In the previous topic's example, you saw that the tinyint data type was implicitly converted to int in the query:

Implicit Conversion

```
DECLARE @myTinyInt AS TINYINT = 25;
DECLARE @myInt as INT = 9999;
SELECT @myTinyInt + @myInt;
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

You might also anticipate that an implicit conversion will take place in the following example:

Implicit Conversion Example

```
DECLARE @somechar CHAR(5) = '6';
DECLARE @someint INT = 1
SELECT @somechar + @someint;
```

Question: Which data type will be converted? To which type?

As you have learned, SQL Server will automatically attempt to perform an implicit conversion from a lower-precedence data type to a higher-precedence.

This is transparent to the user, unless it fails, as in the following example:

Failed Conversion

```
DECLARE @somechar CHAR(3) = 'six';
DECLARE @someint INT = 1
SELECT @somechar + @someint;
```

Returns:

```
Msg 245, Level 16, State 1, Line 3
Conversion failed when converting the varchar value 'six' to data type int.
```

Question: Why does SQL Server attempt to convert the character variable to an integer and not the other way around?

To force SQL Server to convert the int data type to a character for the purposes of the query, you will need to explicitly convert it. You will learn how to do this in the next module.

To learn more about data type conversions, go to Books Online at:

Data Type Conversion (Database Engine)

http://go.microsoft.com/fwlink/?LinkID=402728

Demonstration: SQL Server Data Types

In this demonstration, you will see how to:

- Convert data types.

Demonstration Steps

Convert Data Types

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod06\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod06\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.

7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

Categorize Activity

Categorize each data type into the appropriate category. Indicate your answer by writing the category number to the right of each item.

Items	
1	binary
2	varbinary
3	tinyint
4	decimal
5	int
6	real
7	bigint
8	bit
9	float
10	money

Category 1		Category 2		Category 3
Exact Numeric Data Types		Approximate Numeric Data Types		Binary Data Types

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Working with Character Data

It is likely that much of the data you will work with in your T-SQL queries will involve character data. As you will learn in this lesson, character data involves not only choices of capacity and storage, but also text-specific issues such as language, sort order, and collation. In this lesson, you will learn about the SQL Server character-based data types, how character comparisons work, and some common functions you may find useful in your queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the data types used to store date and time information.
- Enter dates and times as literal values for SQL Server to convert to date and time types.
- Write queries comparing dates and times.
- Write queries using built-in functions to manipulate dates and extract date parts.

Character Data Types

Even though there are many numeric data types in SQL Server, working with numbers is relatively straightforward because there are only so many to work with. By comparison, character data in SQL Server is more complicated, due to issues such as language, character sets, accented characters, sort rules, and case-sensitivity, as well as capacity and storage. Each of these issues will have an impact on which character data type you encounter when writing queries.

Note: Character data is delimited with single quotes.

- One initial choice is character types based on a simple ASCII set versus Unicode, the double-byte character set. Regular, or non-Unicode, characters are limited to a 256-character set and occupy 1 byte per character. These include the CHAR (fixed width) and VARCHAR (varying width) data types. Characters using these data types are delimited with single quotes, such as 'SQL'.
- Unicode data types include NCHAR (fixed width) and NVARCHAR (varying width). These may represent approximately 65,000 different characters—including special characters from many languages—and consume 2 bytes per character. Character strings using this type have an N prefix (for National), such as N'SQL'.
- Character data types also provide for larger storage, in the form of regular and Unicode varying width types declared with the MAX option: VARCHAR(MAX) and NVARCHAR(MAX). These can store up to 2 GB (with each Unicode character using 2 bytes) per instance, and replace the deprecated TEXT and NTEXT data types, respectively.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Character data type ranges and storage requirements are listed in the following table:

Data Type	Range	Storage
CHAR(n), NCHAR(n)	1-8000 characters	n bytes, padded 2*n bytes, padded
VARCHAR(n), NVARCHAR(n)	1-8000 characters	n+2 bytes (2*n) + 2 bytes
VARCHAR(max), NVARCHAR(max)	1-2^31-1 characters	Actual length + 2

Collation

In addition to size and character set, SQL Server character data types are assigned a collation. This assignment may be at one of several levels—the server instance, the database (default), or a collation assigned to a column in a table or expression. Collations are collections of properties that govern several aspects of character data:

- Supported languages
- Sort order
- Case sensitivity
- Accent sensitivity

Note: A default collation is established during the installation of SQL Server but can be overridden on a per-database or per-column basis. As you will see, you may also override the current collation for some character data by explicitly setting a different collation in your query.

When querying, it is important to be aware of the collation settings for your character data. For example, is it case-sensitive? The following query will execute differently, based on whether the column being compared is case-sensitive or not.

If the column is case-sensitive and the desired value is Funk, then this will succeed:

Case-Sensitivity Example

```
SELECT empid, lastname
FROM HR.employees
WHERE lastname = N'Funk';
```

For the same data, this query would return invalid results if the column were case-sensitive:

Case-Sensitivity Example

```
SELECT empid, lastname
FROM HR.employees
WHERE lastname = N'funk';
```

To control how your query is treating collation settings, you can add the optional COLLATE clause to the WHERE clause.

This example will force a case-sensitive and accent-sensitive comparison using the Latin1_General character set:

COLLATE Clause

```
SELECT empid, lastname
FROM HR.employees
WHERE lastname COLLATE Latin1_General_CS_AS = N'Funk';
```

String Concatenation

To concatenate, or join together, two strings, SQL Server uses the + (plus) operator.

The following example concatenates a given name, space, and family name into a single string:

String Concatenation Example

```
SELECT
empid, lastname, firstname, firstname + N'
' + lastname AS fullname
FROM HR.Employees;
```

Note: Since the plus sign is also used for arithmetic addition, consider whether any of your data is numeric when concatenating. Characters have a lower precedence than numbers, and SQL Server will attempt to convert and add mixed data types rather than concatenating them.

SQL Server 2012 introduced a new CONCAT function, which returns a string that is the result of concatenating one or more string values. Unlike the + operator, CONCAT will convert any NULLs to empty strings before concatenation.

The syntax is as follows:

CONCAT Function

```
CONCAT(string_value1, string_value2, string_valueN)
```

An example of CONCAT is:

CONCAT Example

```
SELECT custid, city, region, country,
CONCAT(city, ', ' + region, ', ' + country) AS location
FROM Sales.Customers;
```

This query returns the following partial results:

```
custid city        region country  location
------ ----------- ------ -------- -------------------
1      Berlin      NULL   Germany  Berlin, Germany
2      México D.F. NULL   Mexico   México D.F., Mexico
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
3      México D.F. NULL   Mexico   México D.F., Mexico
4      London      NULL   UK       London, UK
5      Luleå       NULL   Sweden   Luleå, Sweden
```

Go to the topic *CONCAT (Transact-SQL)* in Books Online at:

CONCAT (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402729

Character String Functions

In addition to retrieving character data as-is from SQL Server, you may also need to extract portions of text or determine the location of characters within a larger string. SQL Server provides a number of built-in functions to accomplish these tasks. Some of these functions include:

- Common functions that modify character

Function	Syntax	Remarks
SUBSTRING()	SUBSTRING (expression , start , length)	Returns part of an expression.
LEFT(), RIGHT()	LEFT (expression , integer_value) RIGHT (expression , integer_value)	LEFT() returns left part of string up to integer_value. RIGHT() returns right part of string.
LEN(), DATALENGTH()	LEN (string_expression) DATALENGTH (expression)	LEN() returns the number of characters of the specified string expression, excluding trailing blanks. DATALENGTH() returns the number of bytes used.
CHARINDEX()	CHARINDEX (expressionToFind, expressionToSearch)	Searches an expression for another expression and returns its starting position if found. Optional start position.
REPLACE()	REPLACE (string_expression , string_pattern , string_replacement)	Replaces all occurrences of a specified string value with another string value.
UPPER(), LOWER()	UPPER (character_expression) LOWER (character_expression)	UPPER() returns a character expression with lowercase character data converted to uppercase. LOWER() converts uppercase to lowercase.

- **FORMAT**() – allows you to format an input value to a character string based on a .NET format string, with an optional culture parameter:

This example:

FORMAT Function

```
SELECT top (3) orderid, FORMAT(orderdate,'d','en-us') AS us,
FORMAT(orderdate,'d','de-DE') AS de
FROM Sales.Orders;
```

Returns:

```
Ordered      us           de
-------      --------     ----------
10248        7/4/2006     04.07.2006
10249        7/5/2006     05.07.2006
10250        7/8/2006     08.07.2006
```

- **SUBSTRING()** – for returning part of a character string given a starting point and a number of characters to return.

This example:

SUBSTRING Example

```
SELECT SUBSTRING('Microsoft SQL Server',11,3) AS Result;
```

Returns:

```
Result
------
SQL
```

- **LEFT()** and **RIGHT()** – for returning the leftmost or rightmost characters, respectively, up to a provided point in a string.

MCT USE ONLY STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

See the following example:

LEFT Example

```
SELECT LEFT('Microsoft SQL Server',9) AS Result;
```

Returns:

```
Result
---------
Microsoft
```

- **LEN()** and **DATALENGTH()** – for providing metadata about the number of characters or bytes stored in a string. Given a string padded with spaces.

See the following example:

LEN and DATALENGTH Example

```
SELECT LEN('Microsoft SQL Server     ')            AS [LEN];
SELECT DATALENGTH('Microsoft SQL Server     ')     AS [DATALEN];
```

Returns:

```
LEN
-----------
20
DATALEN
-----------
25
```

- **CHARINDEX()** – for returning a number representing the position of a string within another string.

See the following example:

CHARINDEX Example

```
SELECT CHARINDEX('SQL','Microsoft SQL Server') AS Result;
```

Returns:

```
Result
-----------
11
```

- **REPLACE()** – for substituting one set of characters with another set within a string.

See the following example:

REPLACE Example

```
SELECT REPLACE('Microsoft SQL Server Hekaton','Hekaton','2014 In-Memory OLTP Engine') AS
Result;
```

Returns:

```
Result
------
Microsoft SQL Server 2014 In-Memory OLTP Engine
```

MCT USE ONLY. STUDENT USE PROHIBITED

- **UPPER()** and **LOWER()** – for performing case conversions.

See the following example:

UPPER and LOWER Example

```
SELECT UPPER('Microsoft SQL Server') AS [UP],LOWER('Microsoft SQL Server') AS [LOW];
```

Returns:

```
UP                   LOW
-------------------- --------------------
MICROSOFT SQL SERVER microsoft sql server
```

For references on these and other string functions, go to Books Online at:

String Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402730

The LIKE Predicate

Character-based data in SQL Server provides for more than exact matches in your queries. Through the use of the LIKE predicate, you can also perform pattern matching in your WHERE clause.

The LIKE predicate allows you to check a character string against a pattern. Patterns are expressed with symbols, which can be used alone or in combinations to search within your strings:

- **% (Percent)** represents a string of any length. For example, LIKE N'Sand%' will match 'Sand', 'Sandwich', 'Sandwiches', and so on.
- **_ (Underscore)** represents a single character. For example, LIKE N'_a' will match any string whose second character is an 'a'.
- **[<List of characters>]** represents a single character within the supplied list. For example, LIKE N'[DEF]%' will find any string that starts with a 'D', an 'E', or an 'F'.
- **[<Character> - <character>]** represents a single character within the specified range. For example, LIKE N'[N-Z]%' will match any string that starts with any letter of the alphabet between N and Z, inclusive.
- **[^<Character list or range>]** represents a single character not in the specified list or range. For example, LIKE N'^[A]% ' will match a string beginning with anything other than an 'A'.
- **ESCAPE Character** allows you to search for a character that is also a wildcard character. For example, LIKE N'10% off%' ESCAPE '%' will find any string that starts with 10%, including the literal character '%'.

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Working with Character Data

In this demonstration, you will see how to:

- Manipulate character data.

Demonstration Steps

Manipulate Character Data

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod06\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod06\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

   Question: You have the following query

   ```
   SELECT FirstName
   FROM HumanResources.Employees
   WHERE FirstName LIKE N'^[MA]%'
   ```

   Will the query return an employee with the first name “Matthew”

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Working with Date and Time Data

Date and time data is very common in working with SQL Server data types. In this lesson, you will learn which data types are used to store temporal data, how to enter dates and times so they will be properly parsed by SQL Server, and how to manipulate dates and times with built-in functions.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the data types used to store date and time information.
- Enter dates and times as literal values for SQL Server to convert to date and time types.
- Write queries comparing dates and times.
- Write queries using built-in functions to manipulate dates and extract date parts.

Date and Time Data Types

There has been a progression in SQL Server's handling of temporal data as newer versions are released. Since you may need to work with data created for older versions of SQL Server, even though you're writing queries for SQL Server 2014, it will be useful to review past support for date and time data:

- Older versions of SQL Server supported only DATETIME and SMALLDATETIME
- DATE, TIME, DATETIME2, and DATETIMEOFFSET introduced in SQL Server 2008
- SQL Server 2012 added further functionality for working with date and time data types

Data Type	Storage (bytes)	Date Range	Accuracy	Recommended Entry Format
DATETIME	8	January 1, 1753 to December 31, 9999	3-1/3 milliseconds	'YYMMDD hh:mm:ss.nnn'
SMALLDATETIME	4	January 1, 1900 to June 6, 2079	1 minute	'YYMMDD hh:mm:ss.nnn'
DATETIME2	6 to 8	January 1, 0001 to December 31, 9999	100 nanoseconds	'YYMMDD hh:mm:ss.nnnnnnn'
DATE	3	January 1, 0001 to December 31, 9999	1 day	'YYYY-MM-DD'
TIME	3 to 5		100 nanoseconds	'hh:mm:ss.nnnnnnn'
DATETIMEOFFSET	8 to 10	January 1, 0001 to December 31, 9999	100 nanoseconds	'YY-MM-DD hh:mm:ss.nnnnnnn [+|-]hh:mm'

- Prior to SQL Server 2008, there were only two data types for date and time data: DATETIME and SMALLDATETIME. Each of these stored both date and time in a single value. For example, a DATETIME could store '20140212 08:30:00' to represent February 12 2014 at 8:30 am.
- In SQL Server 2008, Microsoft introduced four new data types: DATETIME2, DATE, TIME, and DATETIMEOFFSET. These addressed issues of precision, capacity, time zone tracking, and separating dates from times.
- In SQL Server 2012, Microsoft introduced new functions for working with partial data from date and time data types (such as DATEFROMPARTS) and for performing calculations on dates (such as EOMONTH).

Date and Time Data Types: Literals

To use date and time data in your queries, you will need to be able to represent temporal data in T-SQL. SQL Server doesn't offer a specific option for entering dates and times, so you will use character strings called literals, which are delimited with single quotes. SQL Server will implicitly convert the literals to date and time values. (You may also explicitly convert literals with the T-SQL CAST function, which you will learn about in the next module).

SQL Server can interpret a wide variety of literal formats as dates, but for consistency and to avoid issues with language or nationality interpretation, it is recommended that you use a neutral format such as 'YYYYMMDD'. To represent February 12, 2014, you would use the literal '20140212'.

To use literals in a query, see the following example:

Literals Example

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate = '20070825';
```

Besides 'YYYYMMDD', other language-neutral formats are available to you:

Data Type	Language-Neutral Formats	Examples
DATETIME	'YYYYMMDD hh:mm:ss.nnn' 'YYYY-MM-DDThh:mm:ss.nnn' 'YYYYMMDD'	'20140212 12:30:15.123' '2014-02-12T12:30:15.123' '20140212'
SMALLDATETIME	'YYYYMMDD hh:mm' 'YYYY-MM-DDThh:mm' 'YYYYMMDD'	'20140212 12:30' '2014-02-12T12:30' '20140212'
DATETIME2	'YYYY-MM-DD' 'YYYYMMDD hh:mm:ss.nnnnnnn' 'YYYY-MM-DD hh:mm:ss.nnnnnnn' 'YYYY-MM-DDThh:mm:ss.nnnnnnn' 'YYYYMMDD' 'YYYY-MM-DD'	'2014-02-12' '20140212 12:30:15.1234567' '2014-02-12 12:30:15.1234567' '2014-02-12T12:30:15.1234567' '20140212' '2014-02-12'
DATE	'YYYYMMDD' 'YYYY-MM-DD'	'20140212' '2014-02-12'
TIME	'hh:mm:ss.nnnnnnn'	'12:30:15.1234567'
DATETIMEOFFSET	'YYYYMMDD hh:mm:ss.nnnnnnn [+|-]hh:mm' 'YYYY-MM-DD hh:mm:ss.nnnnnnn [+|-	'20140212 12:30:15.1234567 +02:00' '2014-02-12 12:30:15.1234567

MCT USE ONLY. STUDENT USE PROHIBITED

Data Type	Language-Neutral Formats	Examples
	]hh:mm' 'YYYYMMDD' 'YYYY-MM-DD'	+02:00' '20140212' '2014-02-12'

Working with Date and Time Separately

As you have learned, some SQL Server temporal data types store both date and time together in one value. DATETIME and DATETIME2 combine year, month, day, hour, minute, seconds, and more. DATETIMEOFFSET also adds time zone information to the date and time. The time and date components are optional in combination data types such as DATETIME2. So, when using these data types, you need to be aware of how they behave when provided with only partial data:

- DATETIME, SMALLDATETIME, DATETIME2, and DATETIMEOFFSET include both date and time data
- If only date is specified, time set to midnight (all zeroes)
- If only time is specified, date set to base date (January 1, 1900)

```
DECLARE @DateOnly DATETIME = '20120212';
SELECT @DateOnly;
```

```
RESULT
-----------------------
2012-02-12 00:00:00.000
```

- If only the date is provided, the time portion of the data type is filled with zeros and the time is considered to be at midnight.

For example, the query:

DATETIME With No Time

```
DECLARE @DateOnly AS DATETIME = '20140212';
SELECT @DateOnly AS RESULT;
```

Returns:

```
RESULT
-----------------------
2014-02-12 00:00:00.000
```

- If no date data is available and you need to store time data in a combination data type, you can enter a "base" date of January 1, 1900. Alternatively, you can use the CAST() function to convert the time data to a combination data type while entering just the time value. SQL Server will assume the base date. Explicit zeros for the date portion are not permitted.

MCT USE ONLY. STUDENT USE PROHIBITED

Querying Date and Time Values

When querying date and time data types, it is important to know whether your source data includes time values other than zeros.

If all your time values are midnight, queries such as the following will work as expected:

Midnight Time Values

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate= '20070825'
```

This query returns:

```
orderid     custid      empid       orderdate
----------- ----------- ----------- -----------------------
10643       1           6           2007-08-25 00:00:00.000
10644       88          3           2007-08-25 00:00:00.000
```

Note that the orderdate time values are all set to zero. This matches the query predicate, which also omits time, implicitly asking only for rows at midnight.

If your data includes time values, you will need to modify the logic to catch time values after midnight. For example, if the following rows existed in an orders2 table:

```
orderid     custid      empid       orderdate
----------- ----------- ----------- -----------------------
10643       1           6           2007-08-29 08:30:00.000
10644       88          3           2007-08-29 11:55:00.000
```

The following query would fail to select them:

See the following example:

Missing Records That Include Time

```
SELECT orderid, empid, custid, orderdate
FROM orders2
WHERE orderdate = '20070829'
```

But this query would successfully retrieve the rows:

See the following example:

Finding Records That Include Time

```
SELECT orderid, empid, custid, orderdate
FROM orders2
WHERE orderdate >= '20070829'
```

Note: The previous example is supplied for illustration only and cannot be run as written in the sample databases supplied with this course.

As a result, you will need to account for time past midnight for rows where there are values stored in the time portion of combination data types.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

Consider the use of range operators instead:

Range Operators

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20070825'
AND   orderdate <  '20070826';
```

Date and Time Functions

Over the years, SQL Server has supplied a number of functions designed to manipulate date and time data. SQL Server 2012 also introduced:

- Functions that return current date and time

• Functions that return current date and time

Function	Return Type	Remarks
GETDATE()	datetime	Current date and time. No time zone offset.
GETUTCDATE()	datetime	Current date and time in UTC.
CURRENT_TIMESTAMP()	datetime	Current date and time. No time zone offset. ANSI standard.
SYSDATETIME()	datetime2	Current date and time. No time zone offset.
STSUTCDATETIME()	datetime2	Current date and time in UTC.
SYSDATETIMEOFFSET()	datetimeoffset	Current date and time. Includes time zone offset.

```
SELECT CURRENT_TIMESTAMP();
SELECT SYSUTCDATETIME();
```

  offering you choices between various return types, as well as whether to include or exclude time zone information.
- Functions that return parts of date and time values, enabling you to extract only the portion of a date or time that your query requires. Note that DATENAME() and DATEPART() offer functionality similar to one another. The difference between them is the return type.
- Functions that return date and time typed data from components such as separately supplied year, month, and day. Previous versions required parsing of strings to assemble a literal that looked like a date. These new functions allow you to pass in simple numeric inputs for the functions to convert to the corresponding date and time value. Note that these functions require all their parameters.
- Functions that modify date and time values, including to increment dates, calculate the last day of a month, and alter time zone offset information.
- Functions that examine date and time values, returning metadata or calculations about intervals between input dates.

Demonstration: Working with Date and Time Data

In this demonstration, you will see how to:

- Query date and time values

Demonstration Steps

Query Data and Time Values

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod06\Setup.cmd as an administrator.

2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod06\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

Categorize Activity

Categorize each data type or function into the appropriate category. Indicate your answer by writing the category number to the right of each item.

Items	
1	Time
2	DateTimeOffSet
3	SmallDateTime
4	EOMONTH()
5	DateTime
6	DateTime2
7	DATEFROMPARTS()
8	Date

Category 1		Category 2		Category 3
Present in all versions of SQL Server		Only present in SQL Server 2008 and later		Only present in SQL Server 2012 and later

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Working with SQL Server 2014 Data Types

Scenario

You are a business analyst for Adventure Works who will be writing reports using corporate databases stored in SQL Server. You have been provided with a set of business requirements for data and will write T-SQL queries to retrieve the specified data from the databases. You will need to retrieve and convert character and temporal data into various formats.

Objectives

After completing this lab, you will be able to:

- Write queries that return data and time data.
- Write queries that use data and time functions.
- Write queries that return character data.
- Write queries that use character functions.

Estimated Time: 60 Minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Writing Queries That Return Date and Time Data

Scenario

Before you start using different date and time functions in business scenarios, you have to practice on sample data.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement to Retrieve all Distinct Customers
3. Write a SELECT Statement to Return the Data Type date
4. Write a SELECT Statement that Uses Different Date and Time Functions
5. Observe the Table Provided by the IT Department

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab06\Starter folder as Administrator.

▶ Task 2: Write a SELECT Statement to Retrieve all Distinct Customers

1. Open the project file D:\Labfiles\Lab06\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to return columns that contain:
   - The current date and time. Use the alias currentdatetime.
   - Just the current date. Use the alias currentdate.

- Just the current time. Use the alias currenttime.
- Just the current year. Use the alias currentyear.
- Just the current month number. Use the alias currentmonth.
- Just the current day of month number. Use the alias currentday.
- Just the current week number in the year. Use the alias currentweeknumber.
- The name of the current month based on the currentdatetime column. Use the alias currentmonthname.

3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab06\Solution\52 - Lab Exercise 1 - Task 1 Result.txt. Your results will be different because of the current date and time value.
4. Can you use the alias currentdatetime as the source in the second column calculation (currentdate)? Please explain.

▶ Task 3: Write a SELECT Statement to Return the Data Type date

1. Write December 11, 2011, as a column with a data type of date. Use the different possibilities inside the T-SQL language (cast, convert, specific function, and so on) and use the alias somedate.

▶ Task 4: Write a SELECT Statement that Uses Different Date and Time Functions

1. Write a SELECT statement to return columns that contain:
   - A date and time value that is three months from the current date and time. Use the alias threemonths.
   - Number of days between the current date and the first column (threemonths). Use the alias diffdays.
   - Number of weeks between April 4, 1992, and September 16, 2011. Use the alias diffweeks.
   - First day in the current month based on the current date and time. Use the alias firstday.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab06\Solution\53 - Lab Exercise 1 - Task 3 Result.txt. Some results will be different because of the current date and time value.

▶ Task 5: Observe the Table Provided by the IT Department

1. The IT department has written a T-SQL statement that creates and populates a table named Sales.Somedates.
2. Execute the provided T-SQL statement.
3. Write a SELECT statement against the Sales.Somedates table and retrieve the isitdate column. Add a new column named converteddate with a new date data type value based on the column isitdate. If the isitdate column cannot be converted to a date data type for a specific row, then return a NULL.
4. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab06\Solution\54 - Lab Exercise 1 - Task 4 Result.txt.
5. What is the difference between the SYSDATETIME and CURRENT_TIMESTAMP functions?
6. What is a language-neutral format for the DATE type?

Results: After this exercise, you should be able to retrieve date and time data using T-SQL.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Writing Queries That Use Date and Time Functions

Scenario

The sales department would like to have different reports that focus on data during specific time frames. The sales staff would like to analyze distinct customers, distinct products, and orders placed near the end of the month. You will have to write the SELECT statements using the different date and time functions.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve All Distinct Customers
2. Write a SELECT Statement to Calculate the First and Last Day of the Month
3. Write a SELECT Statement to Retrieve the Orders Placed in the Last Five Days of the Ordered Month
4. Write a SELECT Statement to Retrieve All Distinct Products Sold in the First 10 Weeks of the Year 2007

▶ Task 1: Write a SELECT Statement to Retrieve All Distinct Customers

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT DISTINCT
custid
FROM Sales.Orders
WHERE
YEAR(orderdate) = 2008
AND MONTH(orderdate) = 2;
```

4. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab06\Solution\62 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 2: Write a SELECT Statement to Calculate the First and Last Day of the Month

1. Write a SELECT statement with these columns:
   - Current date and time
   - First date of the current month
   - Last date of the current month
2. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab06\Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve the Orders Placed in the Last Five Days of the Ordered Month

1. Write a SELECT statement against the Sales.Orders table and retrieve the orderid, custid, and orderdate columns. Filter the results to include only orders placed in the last five days of the order month.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab06\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

▶ Task 4: Write a SELECT Statement to Retrieve All Distinct Products Sold in the First 10 Weeks of the Year 2007

1. Write a SELECT statement against the Sales.Orders and Sales.OrderDetails tables and retrieve all the distinct values for the productid column. Filter the results to include only orders placed in the first 10 weeks of the year 2007.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab06\Solution\65 - Lab Exercise 2 - Task 4 Result.txt.

Results: After this exercise, you should know how to use the date and time functions.

Exercise 3: Writing Queries That Return Character Data

Scenario

Members of the marketing department would like to have a more condensed version of a report for when they talk with customers. They want the information that currently exists in two columns displayed in a single column.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Concatenate Two Columns
2. Add an Additional Column and Treat NULL as an Empty String
3. Write a SELECT Statement to Retrieve All Customers Based on the First Character in the Contact Name

▶ Task 1: Write a SELECT Statement to Concatenate Two Columns

1. Open the project file D:\Labfiles\Lab06\Starter\Project\Project.ssmssln and the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement against the Sales.Customers table and retrieve the contactname and city columns. Concatenate both columns so that the new column looks like this:

```
Allen, Michael (city: Berlin)
```

3. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab06\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Add an Additional Column and Treat NULL as an Empty String

1. Copy the T-SQL statement in task 1 and modify it to extend the calculated column with new information from the region column. Treat a NULL in the region column as an empty string for concatenation purposes. When the region is NULL, the modified column should look like this:

```
Allen, Michael (city: Berlin, region: )
```

2. When the region is not NULL, the modified column should look like this:

```
Richardson, Shawn (city: Sao Paulo, region: SP)
```

3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab06\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Write a SELECT Statement to Retrieve All Customers Based on the First Character in the Contact Name

1. Write a SELECT statement to retrieve the contactname and contacttitle columns from the Sales.Customers table. Return only rows where the first character in the contact name is 'A' through 'G'.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab06\Solution\74 - Lab Exercise 3 - Task 3 Result.txt. Notice the number of rows returned.

Results: After this exercise, you should have an understanding of how to concatenate character data.

Exercise 4: Writing Queries That Use Character Functions

Scenario

The marketing department would like to address customers by their first and last names. In the Sales.Customers table, there is only one column named contactname that has both elements separated by a comma. You will have to prepare a report to show the first and last names separately.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement that Uses the SUBSTRING Function
2. Extend the SUBSTRING Function to Retrieve the First Name
3. Write a SELECT Statement to Change the Customer IDs
4. Challenge: Write a SELECT Statement to Return the Number of Character Occurrences

▶ Task 1: Write a SELECT Statement that Uses the SUBSTRING Function

1. Open the project file D:\Labfiles\Lab06\Starter\Project\Project.ssmssln and the T-SQL script 81 - Lab Exercise 4.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the contactname column from the Sales.Customers table. Based on this column, add a calculated column named lastname, which should consist of all the characters before the comma.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab06\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.

▶ Task 2: Extend the SUBSTRING Function to Retrieve the First Name

1. Write a SELECT statement to retrieve the contactname column from the Sales.Customers table and replace the comma in the contact name with an empty string. Based on this column, add a calculated column named firstname, which should consist of all the characters after the comma.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab06\Solution\83 - Lab Exercise 4 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Change the Customer IDs

1. Write a SELECT statement to retrieve the custid column from the Sales.Customers table. Add a new calculated column to create a string representation of the custid as a fixed-width (six characters) customer code prefixed with the letter C and leading zeros. For example, the custid value 1 should look like C00001.

2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab06\Solution\84 - Lab Exercise 4 - Task 3 Result.txt.

▶ Task 4: Challenge: Write a SELECT Statement to Return the Number of Character Occurrences

1. Write a SELECT statement to retrieve the contactname column from the Sales.Customers table. Add a calculated column, which should count the number of occurrences of the character 'a' inside the contact name. (Hint: Use the string functions REPLACE and LEN.) Order the result from highest to lowest occurrence.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab06\Solution\85 - Lab Exercise 4 - Task 4 Result.txt.

Results: After this exercise, you should have an understanding of how to use the character functions.

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways

Review Question(s)

Question: Will SQL Server be able to successfully implicitly convert an int data type to a varchar?

Question: What data type is suitable for storing flag information, such as TRUE or FALSE?

Question: What logical operators are useful for retrieving ranges of date and time values?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 7

Using DML to Modify Data

Contents:

Module Overview	**7-1**
Lesson 1: Adding Data to Tables	**7-2**
Lesson 2: Modifying and Removing Data	**7-6**
Lesson 3: Generating Numbers	**7-11**
Lab: Using DML to Modify Data	**7-14**
Module Review and Takeaways	**7-16**

Module Overview

Transact-SQL (T-SQL) data manipulation language (DML) includes commands to add and modify data. In this module, you will learn the basics of using INSERT to add rows to tables, UPDATE and MERGE to change existing rows, and DELETE and TRUNCATE TABLE to remove rows. You will also learn how to generate sequences of numbers using the IDENTITY property of a column, as well as the sequence object.

Objectives

After completing this module, you will be able to:

- Write T-SQL statements that insert rows into tables.
- Write T-SQL statements that modify or remove existing rows.
- Write T-SQL statements that automatically generate values for columns.

Lesson 1
Adding Data to Tables

In this lesson, you will learn how to write queries that add new rows to tables.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that use the INSERT statement to add data to tables.
- Use the INSERT statement with SELECT and EXEC clauses.
- Use SELECT INTO to create and populate tables.
- Describe the behavior of default constraints when rows are inserted into a table.

Using INSERT to Add Data

The INSERT statement is used in T-SQL to add one or more rows to a table. There are several forms of the statement.

Its basic syntax appears below:

INSERT Syntax

```
INSERT [INTO] <table or view>
[(column_list)]
VALUES (expression|DEFAULT|NULL, ...n);
```

This form, called INSERT VALUES, allows you to specify the columns to insert and the order in which to insert them. In addition, you can provide the values for those columns. The following example shows the use of the INSERT VALUES statement:

Note the correlation between the columns and the value list:

INSERT VALUES Example

```
INSERT INTO Sales.OrderDetails(orderid, productid, unitprice, qty, discount)
VALUES(12000,39,18,2,0.05);
```

If the column list is omitted, then values (or the DEFAULT or NULL placeholders) must be specified for all columns, in the order in which they are defined in the table. If a value is not specified for a column that does not have a value automatically assigned (such as through a DEFAULT constraint), the INSERT statement will fail.

In addition to inserting a single row at a time, the INSERT VALUES statement can be used to insert multiple rows. Microsoft SQL Server 2008 and later versions support the use of the VALUES clause to build a virtual table, called a table value constructor, made up of multiple rows, called row value constructors. Separate each row value constructor with commas.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The following example inserts multiple rows with a single INSERT statement:

Inserting Multiple Rows

```
INSERT INTO Sales.OrderDetails(orderid, productid, unitprice, qty, discount)
VALUES
      (12001,39,18,2,0.05),
      (12002,39,18,5,0.10);
```

Go to *INSERT (Transact-SQL)* in Books Online at:

INSERT (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402734

Go to *Table Value Constructor (Transact-SQL)* in Books Online at

Table Value Constructor (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402735

Using INSERT with SELECT and EXEC

Beyond specifying a literal set of values in an INSERT statement, T-SQL also supports using the output of other operations to provide values for INSERT. You may pass the results of a SELECT clause or the output of a stored procedure to the INSERT clause.

- INSERT...SELECT is used to insert the result set of a query into an existing table

```
INSERT INTO Sales.OrderHist(
        orderid,custid,empid,orderdate)
SELECT orderid,custid,empid,orderdate
FROM Sales.Orders
WHERE orderdate < '20080101';
```

- INSERT...EXEC is used to insert the result of a stored procedure or dynamic SQL expression into an existing table

```
INSERT INTO dbo.T1 (productid, productname, unitprice)
EXEC Production.ProdsByCategory
        @numrows = 5, @catid=1;
```

To use the SELECT statement with an INSERT statement, build a SELECT clause in the place of the VALUES clause. This form, called INSERT SELECT, allows you to insert the set of rows returned by a SELECT query into a destination table. The use of INSERT SELECT presents the same considerations as INSERT VALUES:

- You may optionally specify a column list following the table name.
- You must provide values, DEFAULT, or NULL placeholders for all columns that do not have values otherwise automatically assigned.

The following syntax illustrates the use of INSERT SELECT:

INSERT SELECT

```
INSERT [INTO] <table or view> [(column_list)]
SELECT <column_list> FROM <table_list>...;
```

Result sets from stored procedures (or even dynamic batches) may also be used as input to an INSERT statement. This form of INSERT, called INSERT EXEC, is conceptually similar to INSERT SELECT and will present the same considerations.

The following example shows the use of an EXEC clause to insert rows from a stored procedure:

Inserting Rows From a Stored Procedure

```
INSERT INTO Production.Products (productname, supplierid, categoryid, unitprice)
EXEC Production.AddNewProducts;
GO
```

Note: The example above references a procedure that is not supplied with the course database. Code to create it appears in the demonstration for this module.

Using SELECT INTO

T-SQL provides the SELECT INTO statement, which allows you to create and populate a new table with the results of a SELECT query. SELECT INTO cannot be used to insert rows into an existing table. A new table is created, with a schema defined by the columns in the SELECT list. Each column in the new table will have the same name, data type, and nullability as the corresponding column (or expression) in the SELECT list.

- SELECT...INTO is similar to INSERT...SELECT but SELECT...INTO creates a new table each time the statement is executed
- Copies column names, data types, and nullability
- Does not copy constraints or indexes

```
SELECT orderid, custid, empid, orderdate, shippeddate
INTO Sales.OrderArchive
FROM Sales.Orders
WHERE orderdate < '20080101';
```

To use SELECT INTO, add INTO <new_target_table_name> in the SELECT clause of the query, just before the FROM clause.

INTO Clause (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402737

To use SELECT INTO, add INTO <new_target_table_name> in the SELECT clause of the query, just before the FROM clause.

See the following example:

SELECT INTO

```
SELECT orderid, custid, empid, orderdate, shipcity, shipregion, shipcountry
INTO Sales.OrdersExport
FROM Sales.Orders
WHERE empid = 5;
```

The results:

```
(42 row(s) affected)
```

Note: The use of SELECT INTO requires CREATE TABLE permission in the destination database.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Inserting Data Into Tables

In this demonstration, you will see how to:

- Insert rows into tables.

Demonstration Steps

Insert Rows Into Tables

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod07\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod07\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Question	
You want to populate three columns of an existing table with data from another table in the same database. Which of the following types of query should you use?	
Select the correct answer.	
	INSERT… VALUES
	INSERT… SELECT
	INSERT… EXEC
	SELECT INTO

Lesson 2
Modifying and Removing Data

In this lesson, you will learn how to write queries that modify or remove rows from a target table. You will also learn how to perform an upsert, in which new rows are added and existing rows are modified in the same operation.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that modify existing rows using UPDATE.
- Write queries that modify existing rows and insert new rows using MERGE.
- Write queries that remove existing rows using DELETE.
- Remove all rows from a table using TRUNCATE.

Using UPDATE to Modify Data

SQL Server provides the UPDATE statement to change existing data in a table or a view. UPDATE operates on a set of rows defined by a condition in a WHERE clause or defined in a join. It uses a SET clause that can perform one or more assignments, separated by commas, to allocate new values to the target. The WHERE clause in an UPDATE statement has the same structure as a WHERE clause in a SELECT statement.

- Updates all rows in a table or view
  - Set can be filtered with a WHERE clause
  - Set can be defined with a JOIN clause
- Only columns specified in the SET clause are modified

```
UPDATE Production.Products
   SET unitprice = (unitprice * 1.04)
WHERE categoryid =  1 AND discontinued = 0;
```

Note: An UPDATE without a WHERE clause or join to filter the set will modify all rows in the target. Use caution!

The following code shows the basic syntax of the UPDATE statement:

UPDATE Syntax

```
UPDATE <table_name>
    SET
        <column_name> = { expression | DEFAULT | NULL }
       [ ,...n ]
```

Any column omitted from the SET clause will not be modified by the UPDATE statement.

The following example uses the UPDATE statement to increase the price of all current products in category 1 from the Production.Products table:

UPDATE Example

```
UPDATE Production.Products
   SET unitprice = (unitprice * 1.04)
WHERE categoryid =  1 AND discontinued = 0;
```

MCT USE ONLY. STUDENT USE PROHIBITED

T-SQL supports compound assignment operators. In the above example, the unitprice value can be assigned as follows:

Compound Assignment Operators

```
SET unitprice *= 1.04
```

Go to *UPDATE (Transact-SQL)* in Books Online at:

UPDATE (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402738

Using MERGE to Modify Data

In database operations, there is a common need to perform an upsert, in which some rows are updated and new rows are inserted from a source set. SQL Server versions prior to 2008, when support for the MERGE statement was added, required multiple operations to update and insert into a target table. The MERGE statement allows you to insert, update, and even delete rows from a target table, based on a join to a source data set, all in a single statement.

- MERGE modifies data based on a condition
  - When the source matches the target
  - When the source has no match in the target
  - When the target has no match in the source

```
MERGE INTO schema_name.table_name AS TargetTbl
   USING (SELECT <select_list>) AS SourceTbl
   ON (TargetTbl.col1 = SourceTbl.col1)
   WHEN MATCHED THEN
      UPDATE SET col2 = SourceTbl.col2
WHEN NOT MATCHED THEN
   INSERT (<column_list>)
   VALUES (<value_list>);
```

MERGE modifies data, based on one or more conditions:

- When the source data matches the data in the target.
- When the source data has no match in the target.
- When the target data has no match in the source.

Note: Because the T-SQL implementation of MERGE supports the WHEN NOT MATCHED BY SOURCE clause, MERGE is more than just an upsert operation.

The following code shows the general syntax of a MERGE statement:

An update is performed on the matching rows when rows are matched between the source and target. An insert is performed when no rows to match the source are found in the target:

MERGE Example

```
MERGE INTO schema_name.table_name AS TargetTbl
    USING (SELECT <select_list>) AS SourceTbl
    ON (TargetTbl.col1 = SourceTbl.col1)
    WHEN MATCHED THEN
        UPDATE SET col2 = SourceTbl.col2
      WHEN NOT MATCHED THEN
    INSERT (<column_list>)
    VALUES (<value_list>);
```

MCT USE ONLY. STUDENT USE PROHIBITED

The following example shows the use of a MERGE statement to update shipping information for existing orders, or to insert rows for new orders when no match is found. Note that this example is for illustration only and cannot be run using the sample database for this course.

See the following example:

MERGE Example

```
MERGE INTO Sales.Orders AS T
      USING Sales.Staging AS S
      ON (T.orderid = S.orderid)
WHEN MATCHED THEN
      UPDATE SET
          T.shippeddate = S.shippeddate,
              T.shipperid = S.shipperid,
              T.freight = S.freight
WHEN NOT MATCHED THEN
      INSERT (orderid,custid,empid,orderdate,shippeddate, shipperid,freight)
      VALUES (
      s.orderid,s.custid,s.empid,s.orderdate,s.shippeddate, s.shipperid, s.freight);
```

Go to *MERGE (Transact-SQL)* in Books Online at:

MERGE (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402739

Using DELETE to Remove Data

SQL Server provides two methods for removing rows from a table: DELETE and TRUNCATE TABLE. The DELETE statement removes all rows from the target table that meet the condition defined in a WHERE clause. If no WHERE clause is specified, all rows in the table will be removed. The WHERE clause in a DELETE statement has the same structure as a WHERE clause in a SELECT or UPDATE statement.

Note: Be careful when using a DELETE statement without a WHERE clause! All rows will be deleted.

The following example uses a DELETE statement without a WHERE clause to remove all rows from a table:

Remove All Rows

```
DELETE FROM dbo.Nums;
```

The following example uses a DELETE statement with a WHERE clause to specify the set of rows to be removed:

DELETE with WHERE

```
DELETE FROM Sales.OrderDetails
WHERE orderid = 10248;
```

MCT USE ONLY STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

SQL Server also supports the use of joins in a DELETE statement, allowing you to use values from another table to specify the set of rows to be removed from the target table. For more information, go to *DELETE (Transact-SQL)* in Books Online at:

DELETE (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402740

Using TRUNCATE TABLE to Remove Data

The TRUNCATE TABLE command removes all rows from a table. Conceptually, it can be thought of as a DELETE statement without a WHERE clause. However, TRUNCATE TABLE differs from a DELETE statement in the following ways:

- TRUNCATE TABLE always removes all rows and does not support a WHERE clause to restrict which rows are deleted.
- TRUNCATE TABLE uses less space in the transaction log than DELETE, since DELETE logs individual row deletions, while TRUNCATE TABLE only logs the deallocation of storage space. As a result, TRUNCATE TABLE can be faster than an unrestricted DELETE for large volumes of data.
- TRUNCATE TABLE cannot be used on a table with a foreign key reference to another.
- If the table contains an IDENTITY column, the counter for that column is reset to the initial seed value defined for the column or a default value of 1. (See the next lesson for more information on IDENTITY.)
- Although a TRUNCATE TABLE operation is said to be minimally logged, it can be rolled back and all rows restored if TRUNCATE is issued within a user-defined transaction.

TRUNCATE TABLE Example

```
TRUNCATE TABLE dbo.Nums;
```

Demonstration: Modifying and Removing Data From Tables

In this demonstration, you will see how to:

- Update and delete data in a table.

Demonstration Steps

Update and Delete Data in a Table

1. On the virtual machine, on the Taskbar, click Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod07\Setup.cmd as an administrator.

2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod07\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

Question: A DBA has accidentally deleted all records in the Cars table by using a DELETE statement. His query was intended to remove all pool cars that have been sold. The query he used was:

```
DELETE FROM Scheduling.Cars
WHERE Cars.DateSold <> NULL
```

What mistake did the DBA make?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Generating Numbers

In this lesson, you will learn how to automatically generate a sequence of numbers for use as column values.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how to use the IDENTITY property of a column to generate a sequence of numbers.
- Describe how to use the sequence object in SQL Server 2012 to generate a sequence of numbers.

Using IDENTITY

You may need to automatically generate sequential values for a column in a table. SQL Server provides two mechanisms for generating values: the IDENTITY property, for all versions of SQL Server, and the sequence object in SQL Server 2012 and SQL Server 2014. Each mechanism can be used to generate sequential numbers when rows are inserted into a table.

- IDENTITY property of a column generates sequential numbers automatically for insertion into a table
  - Can specify optional seed and increment values
- Only one column in a table may have IDENTITY property defined
  - IDENTITY column omitted in INSERT statements
- Functions provided to return last generated values

```
CREATE TABLE Production.Products(
productid int IDENTITY(1,1) NOT NULL,
productname nvarchar(40) NOT NULL,
categoryid int NOT NULL,
unitprice money NOT NULL)
```

To use the IDENTITY property, define a column using a numeric data type with a scale of 0 and include the IDENTITY keyword, an optional seed (starting value), and an increment value (step value). Only one column in a table may have the IDENTITY property set.

Note: IDENTITY values are generated separately for each table that contains an IDENTITY column. However, values are not unique across multiple tables.

The following code fragment shows an Employeeid column defined with the IDENTITY property, a seed of 1000, and an increment of 1:

IDENTITY Example

```
Employeeid int IDENTITY(1000,1) NOT NULL,
```

When an IDENTITY property is defined on a column, INSERT statements against the table do not reference the IDENTITY column. SQL Server will generate a value using the next available value within the column. If a value must be explicitly assigned to an IDENTITY column, the SET IDENTITY INSERT statement must be executed to override the default behavior of the IDENTITY column.

Go to *SET IDENTITY_INSERT (Transact-SQL)* in Books Online at:

SET IDENTITY_INSERT (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402741

Once a value is assigned to a column by the IDENTITY property, the value may be retrieved like any other value in a column. Values generated by the IDENTITY property are unique within a table. However, without a constraint on the column (such as a PRIMARY KEY or UNIQUE constraint), uniqueness is not enforced after the value has been generated.

To return the most recently assigned value within the same session and scope, such as a stored procedure, use the SCOPE_IDENTITY() function. The legacy @@IDENTITY function will return the last value generated during a session, but it does not distinguish scope. Use SCOPE_IDENTITY() for most purposes.

To reset the IDENTITY property by assigning a new seed, use the DBCC CHECKIDENT statement. Go to *DBCC CHECKIDENT (Transact-SQL)* in Books Online at:

DBCC CHECKIDENT (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402742

Using Sequences

As you have learned, the IDENTITY property may be used to generate a sequence of values for a column within a table. However, the IDENTITY property is not suitable for coordinating values across multiple tables within a database. Database administrators and developers have needed to create tables of numbers manually to provide a pool of sequential values across tables.

- Sequence objects added in SQL Server 2012
- Independent objects in database
  - More flexible than the IDENTITY property
  - Can be used as default value for a column
- Manage with CREATE/ALTER/DROP statements
- Retrieve value with the NEXT VALUE FOR clause

```
-- Define a sequence
CREATE SEQUENCE dbo.InvoiceSeq AS INT START WITH 1
INCREMENT BY 1;
-- Retrieve next available value from sequence
SELECT NEXT VALUE FOR dbo.InvoiceSeq;
```

SQL Server 2012 provides the new sequence object, an independent database object which is more flexible than the IDENTITY property, and can be referenced by multiple tables within a database. The sequence object is created and managed with typical Data Definition Language (DDL) statements such as CREATE, ALTER, and DROP. SQL Server provides a command for retrieving the next value in a sequence, such as within an INSERT statement or a default constraint in a column definition.

To define a sequence, use the CREATE SEQUENCE statement, optionally supplying the data type (must be an integer type or decimal/numeric with a scale of 0), the starting value, an increment value, a maximum value, and other options related to performance.

Go to *CREATE SEQUENCE (Transact-SQL)* in Books Online at:

CREATE SEQUENCE (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402743

To retrieve the next available value from a sequence, use the NEXT VALUE FOR function. To return a range of multiple sequence numbers in one step, use the system procedure sp_sequence_get_range.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The following code defines a sequence and returns an available value to an INSERT statement against a sample table:

SEQUENCE Example

```
CREATE SEQUENCE dbo.demoSequence
    AS INT
    START WITH 1
    INCREMENT BY 1;
GO
CREATE TABLE dbo.tblDemo
     (SeqCol int PRIMARY KEY,
    ItemName nvarchar(25) NOT NULL);
GO
INSERT INTO dbo.tblDemo (SeqCol,ItemName)
    VALUES (NEXT VALUE FOR dbo.demoSequence, 'Item');
GO
```

To inspect the results, query the table:

SELECT * FROM dbo.tblDemo;

The results:

```
SeqCol ItemName
------ --------
Item
```

Check Your Knowledge

Question	
You are using an IDENTITY column to store the order in which orders were placed in a given year. It is a new year and you want to start the count again from 1. Which of the following statements should you use?	
Select the correct answer.	
	OrderSequence int IDENTITY(1,1) NOT NULL
	SET IDENTITY INSERT
	SCOPE_IDENTITY()
	DBCC CHECKIDENT
	CREATE SEQUENCE

Lab: Using DML to Modify Data

Scenario

You are a database developer for Adventure Works and need to create DML statements to update data in the database to support the website development team. The team need T-SQL statements that they can use to carry out updates to data, based on actions performed on the website. You will supply DML statements that they can modify to their specific requirements.

Objectives

After completing this lab, you will be able to:

- Insert records.
- Update and delete records.

Estimated Time: 30 Minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Inserting Records with DML

Scenario

You need to add a new employee to the database and test the required T-SQL code. You can then pass the T-SQL to the human resources system's web developers, who are creating a web form to simplify this task. You also want to add all potential customers to the customers table to consolidate those records.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Insert a Row
3. Insert a Row with SELECT

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab07\Starter folder as Administrator.

▶ Task 2: Insert a Row

1. Open the project file D:\Labfiles\Lab07\Starter\Project\Project.ssmssln and the T-SQL script 41 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.
2. Write an INSERT statement to add a record to the Employees table with the following values:
   - Title: Sales Representative
   - Titleofcourtesy: Mr
   - FirstName: Laurence
   - Lastname: Grider
   - Hiredate: 04/04/2013

MCT USE ONLY. STUDENT USE PROHIBITED

- Birthdate: 10/25/1975
- Address: 1234 1st Ave. S.E.
- City: Seattle
- Country: USA
- Phone: (206)555-0105

▶ Task 3: Insert a Row with SELECT

1. Write an INSERT statement to add all the records from the PotentialCustomers table to the Customers table.

Results: After successfully completing this exercise, you will have one new employee and three new customers.

Exercise 2: Update and Delete Records Using DML

Scenario

You want to update the use of contact titles in the database to match the most commonly-used term in the company and, therefore, make searches more straightforward. You also want to remove the three potential customers that have now been added to the customers table.

The main tasks for this exercise are as follows:

1. Update Rows

2. Delete Rows

▶ Task 1: Update Rows

1. Write an UPDATE statement to update all the records in the **Customers** table which have a city of **Berlin** and a **contacttitle** of **Sales Representative** to have a **contacttitle** of **Sales Consultant**.

▶ Task 2: Delete Rows

1. Write a DELETE statement to delete all the records in the PotentialCustomers table which have the contactname of 'Taylor, Maurice, 'Mallit, Ken', or 'Tiano, Mike', as these records have now been added to the Customers table.

Results: After successfully completing this exercise, you will have updated all the records in the Customers table which have a city of Berlin and a contacttitle of Sales Representative to have a contacttitle of Sales Consultant. You will also have deleted the three records in the PotentialCustomers table, which have already been added to the Customers table.

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways

Review Question(s)

Question: What attributes of the source columns are transferred to a table created with a SELECT INTO query?

Question: The presence of which constraint prevents TRUNCATE TABLE from executing?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 8

Using Built-In Functions

Contents:

Module Overview	8-1
Lesson 1: Writing Queries with Built-In Functions	8-2
Lesson 2: Using Conversion Functions	8-8
Lesson 3: Using Logical Functions	8-13
Lesson 4: Using Functions to Work with NULL	8-17
Lab: Using Built-In Functions	8-21
Module Review and Takeaways	8-25

Module Overview

In addition to retrieving data as it is stored in columns, you will often have to compare or further manipulate values in your T-SQL queries. In this module, you will learn about many functions that are built into Microsoft ® SQL Server®, providing data type conversion, comparison, and NULL handling. You will learn about the various types of functions in SQL Server and how they are categorized. You will work with scalar functions and see where they may be used in your queries. You will learn conversion functions for changing data between different data types and how to write logical tests. You will learn how to work with NULLs and use built-in functions to select non-NULL values, as well as replace certain values with NULL when applicable.

Objectives

After completing this module, you will be able to:

- Write queries with built-in scalar functions.
- Use conversion functions.
- Use logical functions.
- Use functions that work with NULL.

Lesson 1
Writing Queries with Built-In Functions

SQL Server provides many built-in functions, ranging from those that perform data type conversion to those that aggregate and analyze groups of rows. In this lesson, you will learn about the types of functions provided by SQL Server, and then focus on working with scalar functions.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the types of built-in functions provided by SQL Server.
- Write queries using scalar functions.
- Describe aggregate, window, and rowset functions.

SQL Server Built-In Function Types

Functions built into SQL Server can be categorized as follows:

Function Category	Description
Scalar	Operate on a single row, return a single value
Grouped Aggregate	Take one or more input values, return a single summarizing value
Window	Operate on a window (set) of rows
Rowset	Return a virtual table that can be used subsequently in a T-SQL statement

This course will cover grouped aggregates and window functions in later modules, while rowset functions are beyond the scope of the course. The rest of this module will cover various scalar functions.

MCT USE ONLY. STUDENT USE PROHIBITED

Scalar Functions

Scalar functions return a single value. The number of inputs they take may range from zero (such as GETDATE) to one (such as UPPER) to multiple (such as DATEADD). Since they always return a single value, they may be used anywhere a single value (the result) could exist in its own right, from SELECT clauses to WHERE clause predicates.

Built-in scalar functions can be organized into many categories, such as string, conversion, logical, mathematical, and others. This lesson will look at a few common scalar functions.

Some considerations when using scalar functions include:

- Determinism: Will the function return the same value for the same input and database state each time? Many built-in functions are non-deterministic, and as such their results cannot be indexed. This will have an impact on the query processor's ability to use an index when executing the query.
- Collation: When using functions that manipulate character data, which collation will be used? Some functions use the collation of the input value, and others use the collation of the database if no input collation is supplied.

At the time of this writing, Books Online listed more than 200 scalar functions. While this course cannot begin to cover each of them individually, here are some representative examples:

- Date and time functions (covered previously in this course)
- Mathematical functions
- Conversion functions (covered in more detail later in this module)
- System metadata functions

The following example of the YEAR function shows a typical use of a scalar function in a SELECT clause. The function is calculated once per row, using a column from the row as its input:

Scalar Function in a SELECT Clause

```
SELECT orderid, orderdate, YEAR(orderdate) AS orderyear
FROM Sales.Orders;
```

The results:

```
orderid     orderdate               orderyear
----------- ----------------------- -----------
10248       2006-07-04 00:00:00.000 2006
10249       2006-07-05 00:00:00.000 2006
10250       2006-07-08 00:00:00.000 2006
```

MCT USE ONLY. STUDENT USE PROHIBITED

The following example of the mathematical ABS function shows it being used to return an absolute value multiple times in the same SELECT clause, with differing inputs:

Returning an Absolute Value

```
SELECT ABS(-1.0), ABS(0.0), ABS(1.0);
```

The results:

```
--- --- ---
1.0 0.0 1.0
```

The following example uses the system metadata function DB_NAME() to return the name of the database currently in use by the user's session:

Metadata Function

```
SELECT DB_NAME() AS current_database;
```

The results:

```
current_database
----------------
TSQL
```

Additional information about scalar functions and categories can be found in Books Online at:

Built-in Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402744

Aggregate Functions

Grouped aggregate functions operate on sets of rows defined in a GROUP BY clause and return a summarized result. Examples include SUM, MIN, MAX COUNT, and AVG. In the absence of a GROUP BY clause, all rows are considered one set and the aggregation is performed on all of them.

- Functions that operate on sets, or rows of data
- Summarize input rows
- Without GROUP BY clause, all rows are arranged as one group
- Will be covered later in the course

```
SELECT COUNT(*) AS numorderlines,
       SUM(qty*unitprice) AS totalsales
FROM   Sales.OrderDetails;
```

```
numorderlines totalsales
------------- -----------
2155          56500.91
```

The following example uses a COUNT function and a SUM function to return aggregate values without a GROUP BY clause:

Aggregate Functions

```
SELECT COUNT(*) AS numorders,
SUM(unitprice) AS totalsales
FROM   Sales.OrderDetails;
```

The results:

```
numorders   totalsales
----------- ----------
2155        56500.91
```

MCT USE ONLY, STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

Note: Grouped aggregate functions and the GROUP BY clause will be covered in a later module.

Window Functions

Window functions allow you to perform calculations against a user-defined set, or window, of rows. They include ranking, offset, aggregate, and distribution functions. Windows are defined using the OVER clause, and then window functions are applied to the sets defined.

This example uses the RANK function to calculate a ranking based on the unitprice, with the highest price ranked at 1, the next highest ranked 2, and so on:

- Functions applied to a window, or set of rows
- Include ranking, offset, aggregate and distribution functions
- Will be covered later in the course

```
SELECT TOP(5) productid, productname, unitprice,
      RANK() OVER(ORDER BY unitprice DESC) AS
      rankbyprice
FROM Production.Products
ORDER BY rankbyprice;
```

```
productid productname   unitprice rankbyprice
--------- ------------- --------- -----------
8         Product QDOMO 263.50    1
29        Product VJXYN 123.79    2
9         Product AOZBW 97.00     3
20        Product QHFFP 81.00     4
18        Product CKEDC 62.50     5
```

Window Function

```
SELECT TOP(5) productid, productname, unitprice,
      RANK() OVER(ORDER BY unitprice DESC) AS rankbyprice
FROM Production.Products
ORDER BY rankbyprice;
```

The results:

```
productid   productname   unitprice rankbyprice
----------- ------------- --------- -----------
38          Product QDOMO 263.50    1
29          Product VJXYN 123.79    2
9           Product AOZBW 97.00     3
20          Product QHFFP 81.00     4
18          Product CKEDC 62.50     5
```

Note: Window functions will be covered later in this course. This example is provided for illustration only.

Rowset Functions

Rowset functions return a virtual table that can be used elsewhere in the query and take parameters specific to the rowset function itself. They include OPENDATASOURCE, OPENQUERY, OPENROWSET, and OPENXML.

For example, the OPENQUERY function allows you to pass a query to a linked server. It takes the system name of the linked server and the query expression as parameters. The results of the query are returned as a rowset, or virtual table, to the query containing the OPENQUERY function.

- Return an object that can be used like a table in a T-SQL statement
- Include OPENDATASOURCE, OPENQUERY, OPENROWSET, and OPENXML
- Beyond the scope of this course

Further discussion of rowset functions is beyond the scope of this course. For more information, go to Books Online at:

Rowset Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402746

Demonstration: Writing Queries Using Built-In Functions

In this demonstration, you will see how to:

- Use built-in scalar functions.

Demonstration Steps

Use Built-in Scalar Functions

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod08\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod08\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

Categorize Activity

Categorize each function into the appropriate category. Indicate your answer by writing the category number to the right of each function.

Items	
1	MIN()
2	OPENXML()
3	GETDATE()
4	ABS()
5	SUM()
6	AVG()
7	DATEADD()
8	COUNT()
9	OPENDATASOURCE()

MCT USE ONLY. STUDENT USE PROHIBITED

Items	
10	OPENQUERY()
11	OPENROWSET()
12	MAX()
13	UPPER()
14	YEAR()
15	DB_NAME()

Category 1		Category 2		Category 3
Scalar Functions		Aggregate Functions		Rowset Functions

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Using Conversion Functions

When writing T-SQL queries, it's very common to need to convert data between data types. Sometimes the conversion happens automatically, and sometimes you need to control it. In this lesson, you will learn how to explicitly convert data between types using several SQL Server functions. You will also learn to work with functions in SQL Server 2014 that provide additional flexibility during conversion.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the difference between implicit and explicit conversions.
- Describe when you will need to use explicit conversions.
- Explicitly convert between data types using the CAST and CONVERT functions.
- Convert strings to date and numbers with the PARSE, TRY_PARSE, and TRY_CONVERT functions.

Implicit and Explicit Data Type Conversions

Earlier in this course, you learned that there are scenarios when data types may be converted during SQL Server operations. You learned that SQL Server may implicitly convert data types, following the precedence rules for type conversion. However, you may need to override the type precedence, or force a conversion where an implicit conversion might fail.

- Implicit conversion occurs automatically
  - Follows data type precedence rules
- Use explicit conversion
  - When implicit would fail or is not permitted
  - To override data type precedence
- Explicitly convert between types with CAST or CONVERT functions
- Watch for truncation

To accomplish this, you can use the CAST and CONVERT functions, as well as the TRY_CONVERT function.

Some considerations when converting between data types include:

- **Collation**. When CAST or CONVERT returns a character string from a character string input, the output uses the same collation. When converting from a non-character type to a character, the return value uses the collation of the database. The COLLATE option may be used with CAST or CONVERT to override this behavior.
- **Truncation**. When you convert data between character or binary types and different data types, data may be truncated, it might appear cut off, or an error could be thrown because the result is too short to display. Which of these results occurs depends on the data types involved. For example, conversion from an integer with a two-digit value to a char(1) will return an '*' which means the character type was too small to display the results.

Additional reading about truncation behavior can be found in Books Online at:

CAST and CONVERT (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402747

MCT USE ONLY. STUDENT USE PROHIBITED

For more information on data type conversions, go to *Data Type Conversion (Database Engine)* in Books Online at:

Data Type Conversion (Database Engine)

http://go.microsoft.com/fwlink/?LinkID=402728

Converting with CAST

To convert a value from one data type to another, SQL Server provides the CAST function. CAST is an ANSI-standard function and is therefore recommended over the SQL Server-specific CONVERT function, which you will learn about in the next topic.

As CAST is a scalar function, you may use it in SELECT and WHERE clauses.

The syntax is as follows:

- Converts a value from one data type to another
  - Can be used in SELECT and WHERE clauses
  - ANSI standard

CAST Syntax:

```
CAST(<value> AS <datatype>)
```

CAST Example:

```
SELECT CAST(SYSDATETIME() AS date);
```

- Returns an error if data types are incompatible:

```
--attempt to convert datetime2 to int
SELECT CAST(SYSDATETIME() AS int);
```

```
Msg 529, Level 16, State 2, Line 1
Explicit conversion from data type datetime2 to int is not allowed.
```

Converting with CAST

```
CAST(<value> AS <datatype>)
```

The following example from the TSQL sample database uses CAST to convert the orderdate from datetime to date:

CAST Example

```
SELECT orderid, orderdate AS order_datetime, CAST(orderdate AS DATE) AS order_date
FROM Sales.Orders;
```

The results:

```
orderid     order_datetime          order_date
----------- ----------------------- ----------
10248       2006-07-04 00:00:00.000 2006-07-04
10249       2006-07-05 00:00:00.000 2006-07-05
10250       2006-07-08 00:00:00.000 2006-07-08
```

If the data types are incompatible, such as attempting to convert a date to a numeric value, CAST will return an error:

CAST With Incompatible Data Types

```
SELECT CAST(SYSDATETIME() AS int);
```

The results:

```
Msg 529, Level 16, State 2, Line 1
Explicit conversion from data type datetime2 to int is not allowed.
```

For more information about CAST, go to Books Online at:

CAST and CONVERT (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402747

MCT USE ONLY STUDENT USE PROHIBITED

Converting with CONVERT

In addition to CAST, SQL Server provides the CONVERT function. Unlike the ANSI-standard CAST function, the CONVERT function is proprietary to SQL Server and is therefore not recommended. However, because of its additional capability to format the return value, you may still need to use CONVERT occasionally.

- Converts a value from one data type to another
  - Can be used in SELECT and WHERE clauses
  - CONVERT is specific to SQL Server, not standards-based
- Style specifies how input value is converted:
  - Date, time, numeric, XML, and so on

Syntax:

```
CONVERT (<datatype>, <value>, <optional style no.>)
```

Example:

```
CONVERT(CHAR(8), CURRENT_TIMESTAMP,112) AS ISO_style;

ISO_style
---------
20120212
```

As with CAST, CONVERT is a scalar function. You may use CONVERT in SELECT and WHERE clauses.

The syntax is as follows:

Converting with CONVERT

```
CONVERT(<datatype>, <value>, <optional_style_number>);
```

The style number argument causes CONVERT to format the return data according to a specified set of options. These cover a wide range of date and time styles, as well as styles for numeric, XML and binary data. Some date and time examples include:

Style Without Century	Style With Century	Standard Label	Value
1	101	U.S.	mm/dd/yyyy
2	102	ANSI	yy.mm.dd - no change for century
12	112	ISO	yymmdd or yyyymmdd

The following example uses CONVERT to convert the current time from datetime to char(8):

CONVERT Example

```
SELECT CONVERT(CHAR(8), CURRENT_TIMESTAMP, 12) AS ISO_short, CONVERT(CHAR(8),
CURRENT_TIMESTAMP, 112) AS ISO_long;
```

The results:

```
ISO_short ISO_long
--------- --------
120212    20120212
```

For more information about CONVERT and its style options, go to Books Online at:

CAST and CONVERT (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402747

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

Converting Strings with PARSE

A very common business problem is building a date, time, or numeric value from one or more strings, often concatenated. SQL Server 2014 makes this task easier with the PARSE function.

It takes a string, which must be in a form recognizable to SQL Server as a date, time, or numeric value, and returns a value of the specified data type:

- PARSE is a new function in SQL Server 2012
- Converts strings to date, time, and number types

PARSE element	Comment
String_value	Formatted nvarchar(4000) input
Data_type	Requested data type ouput
Culture	Optional string in .NET culture form: en-US, es-ES, ar-SA, and so on

- PARSE example:

```
SELECT PARSE('02/12/2012' AS datetime2
  USING 'en-US') AS parse_result;
```

Converting Strings with PARSE

```
SELECT PARSE('<string_value>',<data_type>
[USING <culture_code>]);
```

The culture parameter must be in the form of a valid .NET Framework culture code, such as 'en-US' for US English, 'es-ES' for Spanish, and so on. If the culture parameter is omitted, the settings for the current user session will be used.

The following example converts the string '02/12/2012' into a datetime2, using the en-US culture codes:

PARSE Example

```
SELECT PARSE('02/12/2012' AS datetime2 USING 'en-US') AS us_result;
```

The results:

```
us_result
----------------------
2012-02-12 00:00:00.00
```

For more information about PARSE, including culture codes, see Books Online at:

PARSE (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402732

Converting with TRY_PARSE and TRY_CONVERT

When using CONVERT or PARSE, an error may occur if the input value cannot be converted to the specified output type.

For example, if February 31, 2012 (an invalid date) is passed to CONVERT, a runtime error is raised:

- TRY_PARSE and TRY_CONVERT:
  - New in SQL Server 2012
  - Return the results of a data type conversion
    - Like PARSE and CONVERT, they convert strings to date, time and numeric types
    - Unlike PARSE and CONVERT, they return a NULL if the conversion fails

TRY_PARSE Example:

```
SELECT TRY_PARSE('SQLServer' AS datetime2
  USING 'en-US') AS try_parse_result;
```

```
try_parse_result
----------------
NULL
```

CONVERT Error

```
SELECT CONVERT(datetime2, '20120231');
```

The result:

```
--Msg 241, Level 16, State 1, Line 1
--Conversion failed when converting date and/or time from character string.
```

SQL Server 2014 provides conversion functions to address this. TRY_PARSE and TRY_CONVERT will attempt a conversion, just like PARSE and CONVERT, respectively. However, instead of raising a runtime error, failed conversions return NULL.

The following examples compare PARSE and TRY_PARSE behavior. First, PARSE attempts to convert an invalid date:

PARSE Error

```
SELECT PARSE('20120231' AS datetime2 USING 'en-US')
```

Returns:

```
--Msg 9819, Level 16, State 1, Line 1
--Error converting string value 'sqlserver' into data type datetime2 using culture
'en-US'.
```

In contrast, TRY_PARSE handles the error more gracefully:

TRY_PARSE Example

```
SELECT TRY_PARSE('20120231' AS datetime2 USING 'en-US')
```

Returns:

```
------------------------
NULL
```

Demonstration: Using Conversion Functions

In this demonstration, you will see how to:

- Use functions to convert data.

Demonstration Steps

Use Functions to Convert Data

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod08\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod08\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

   Question: You are writing a query against a Human Resources database. You want to ensure that the Employee.StartDate values are displayed in standard British form. What function should you use?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Using Logical Functions

So far in this module, you have learned how to use built-in scalar functions to perform data conversions. In this lesson, you will learn how to use logical functions that evaluate an expression and return a scalar result.

Lesson Objectives

After completing this lesson, you will be able to:

- Use T-SQL functions to perform logical functions.
- Perform conditional tests with the IIF function.
- Select items from a list with CHOOSE.

Writing Logical Tests with Functions

A useful function for validating the data type of an expression is ISNUMERIC. This tests an input expression and returns a 1 if the expression is convertible to any numeric type, including integers, decimals, money, floating point, and real. If the value is not convertible to a numeric type, ISNUMERIC returns a 0.

In the following example, which uses the TSQL sample database, any employee with a numeric postal code is returned:

Writing Logical Tests with Functions

```
SELECT empid, lastname, postalcode
FROM HR.Employees
WHERE ISNUMERIC(postalcode)=1;
```

The results:

```
empid       lastname             postalcode
----------- -------------------- ----------
1           Davis                10003
2           Funk                 10001
3           Lew                  10007
4           Peled                10009
5           Buck                 10004
6           Suurs                10005
7           King                 10002
8           Cameron              10006
9           Dolgopyatova         10008
```

Question: How might you use ISNUMERIC when testing data quality?

Performing Conditional Tests with IIF

IIF is a logical function in SQL Server. If you have used Visual Basic for Applications in Microsoft Excel®, used Microsoft Access®, or created expressions in SQL Server Reporting Services, you may have used IIF.

As in those environments, IIF accepts three parameters – a logical test to perform, a value to return if the test evaluates to true, and a value to return if the test evaluates to false or unknown:

- IIF returns one of two values, depending on a logical test
- Shorthand for a two-outcome CASE expression

IIF Element	Comments
Boolean_expression	Logical test evaluating to TRUE, FALSE, or UNKNOWN
True_value	Value returned if expression evaluates to TRUE
False_value	Value returned if expression evaluates to FALSE or UNKNOWN

```
SELECT productid, unitprice,
       IIF(unitprice > 50, 'high','low') AS pricepoint
FROM Production.Products;
```

IIF Syntax

```
SELECT IIF(<Boolean
expression>,<value_if_TRUE>,<value_if_FALSE_or_UNKNOWN);
```

You can think of IIF as a shorthand approach to writing a CASE statement with two possible return values. As with CASE, you may nest an IIF function within another IIF, down to a maximum level of 10.

The following example uses IIF to return a "high" or "low" label for products based on their unitprice:

IIF Example

```
SELECT productid, unitprice,
      IIF(unitprice > 50, 'high','low') AS pricepoint
FROM Production.Products;
```

Returns:

```
productid   unitprice            pricepoint
----------- --------------------- ----------
7           30.00                 low
8           40.00                 low
9           97.00                 high
17          39.00                 low
18          62.50                 high
```

To learn more about this logical function, go to *IIF (Transact-SQL)* in Books Online at:

IIF (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402748

MCT USE ONLY, STUDENT USE PROHIBITED

Selecting Items from a List with CHOOSE

CHOOSE is another logical function in SQL Server. It is similar to the function of the same name in Microsoft Access.

CHOOSE returns an item from a list, selecting the item that matches an index value:

CHOOSE Syntax

```
SELECT CHOOSE(<index_value>,<item1>,
<item2>[,...]);
```

The following example uses CHOOSE to return a category name based on an input value:

CHOOSE Example

```
SELECT CHOOSE (3, 'Beverages', 'Condiments', 'Confections') AS choose_result;
```

Returns:

```
choose_result
-------------
Confections
```

Note: If the index value supplied to CHOOSE does not correspond to a value in the list, CHOOSE will return a NULL.

For additional information about this logical function, go to *CHOOSE (Transact-SQL)* in Books Online at:

CHOOSE (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402749

Demonstration: Using Logical Functions

In this demonstration, you will see how to:

- Use logical functions.

Demonstration Steps

Use Logical Functions

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod08\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod08\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

Question: You have the following query:

```
SELECT e.FirstName, e.LastName, e.FirstAider
FROM Employees AS e
```

The FirstAider column contains ones and zeros. How can you change the query to make the results more readable?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 4
Using Functions to Work with NULL

You will often need to take special steps to deal with NULL. Earlier in this module, you learned how to test for NULL with ISNULL. In this module, you will learn additional functions for working with NULL.

Lesson Objectives

After completing this lesson, you will be able to:

- Use ISNULL to replace NULLs.
- Use the COALESCE function to return non-NULL values.
- Use the NULLIF function to return NULL if values match.

Converting NULL with ISNULL

In addition to data type conversions, SQL Server provides functions for conversion or replacement of NULL. Both COALESCE and ISNULL can replace NULL input with another value.

To use ISNULL, supply an expression to check for NULL and a replacement value, as in the following example using the TSQL sample database:

For customers with a region evaluating to NULL, the literal "N/A" is returned by the ISNULL function in this example:

- ISNULL replaces NULL with a specified value
- Not standard; use COALESCE instead
- Syntax:

ISNULL Element	Comment
expression_to_check	Return expression itself if not NULL
replacement_value	Returned if expression evaluates to NULL

```
SELECT custid, city, ISNULL(region, 'N/A') AS region, country
FROM Sales.Customers;
```

```
custid      city            region          country
----------- --------------- --------------- ---------------
7           Strasbourg      N/A             France
9           Marseille       N/A             France
32          Eugene          OR              USA
43          Walla Walla     WA              USA
45          San Francisco   CA              USA
```

Converting NULL with ISNULL

```
SELECT custid, city, ISNULL(region, 'N/A') AS region, country
FROM Sales.Customers;
```

The result:

```
custid      city            region          country
----------- --------------- --------------- ---------------
40          Versailles      N/A             France
41          Toulouse        N/A             France
43          Walla Walla     WA              USA
45          San Francisco   CA              USA
```

Note: ISNULL is not standard. Use COALESCE instead. COALESCE will be covered later in this module.

To learn more about ISNULL, go to *ISNULL (Transact-SQL)* in Books Online at:

ISNULL (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402750

Using COALESCE to Return Non-NULL Values

- COALESCE returns the first non-NULL value in a list
  - With only two arguments, COALESCE behaves like ISNULL
  - If all arguments are NULL, COALESCE returns NULL
- COALESCE is standards-based
- Example:

```
SELECT custid, country, region, city,
    country + ',' + COALESCE(region, ' ') + ', ' + city as location
FROM Sales.Customers;
```

- Results:

```
custid country region city        location
------ ------- ------ ----------- ----------------------
17     Germany NULL   Aachen      Germany, , Aachen
65     USA     NM     Albuquerque USA,NM, Albuquerque
55     USA     AK     Anchorage   USA,AK, Anchorage
83     Denmark NULL   Århus       Denmark, , Århus
```

Earlier in this module, you learned how to use the ISNULL function to test for NULL. Since ISNULL is not ANSI standard, you may wish to use the COALESCE function instead. COALESCE takes as its input one or more expressions, and returns the first non-NULL argument it finds.

With only two arguments, COALESCE behaves like ISNULL. However, with more than two arguments, COALESCE can be used as an alternative to a multi-part CASE expression using ISNULL.

If all arguments are NULL, COALESCE returns NULL.

The syntax is as follows:

COALESCE Syntax

```
SELECT COALESCE(<expression_1>[, ...<expression_n>];
```

The following example returns customers with regions where available, and adds a new column combining country, region and city, replacing NULL regions with a space:

COALESCE Example

```
SELECT       custid, country, region, city,
             country + ',' + COALESCE(region, ' ') + ', ' + city as location
FROM Sales.Customers;
```

Returns:

```
custid country region city        location
------ ------- ------ ----------- ----------------------
17     Germany  NULL  Aachen      Germany, , Aachen
65     USA      NM    Albuquerque USA,NM, Albuquerque
55     USA      AK    Anchorage   USA,AK, Anchorage
83     Denmark  NULL  Århus       Denmark, , Århus
```

For more information on COALESCE and comparisons to ISNULL, go to Books Online at:

COALESCE (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402751

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Using NULLIF to Return NULL If Values Match

The NULLIF function is the first you will learn in this module that is designed to return NULL if its condition is met. NULLIF returns NULL when two arguments match. This has useful applications in areas such as data cleansing, when you wish to replace blank or placeholder characters with NULL.

NULLIF takes two arguments and returns NULL if they both match. If they are not equal, NULLIF returns the first argument.

In this example, NULLIF replaces an empty string (if present) with a NULL, but returns the employee middle initial if it is present:

NULLIF Example

```
SELECT empid, lastname, firstname, NULLIF(middleinitial,' ') AS middle_initial
FROM HR.Employees;
```

Returns:

```
empid       lastname             firstname  middle_initial
----------- -------------------- ---------- --------------
1           Davis                Sara       NULL
2           Funk                 Don        D
3           Lew                  Judy       NULL
4           Peled                Yael       Y
```

Note: This example is provided for illustration only and will not run against the sample database supplied with this course.

For more information, go to *NULLIF (Transact-SQL)* in Books Online at:

NULLIF (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402752

Demonstration: Using Functions to Work with NULL

In this demonstration, you will see how to:

- Use functions to work with NULL.

Demonstration Steps

Use Functions to Work with NULL

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod08\Setup.cmd as an administrator.

2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod08\Demo folder.
3. In Solution Explorer, open the **41 – Demonstration D.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

.

Check Your Knowledge

Question	
You are writing a query against the Employees table in the Human Resources database. The CurrentStatus column can contain the string values "New", "Retired", and "Under Caution". Many employees have this column set to NULL when those statuses do not apply to them. For confidentiality, you want to ensure that the employees currently under caution are displayed like those employees with no applicable status. What function should you use?	
Select the correct answer.	
	ISNULL()
	COALLESCE()
	NULLIF()
	TRY_PARSE()
	PARSE()

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Using Built-In Functions

Scenario

You are a business analyst for Adventure Works, who will be writing reports using corporate databases stored in SQL Server. You have been provided with a set of business requirements for data and you will write T-SQL queries to retrieve the specified data from the databases. You will need to retrieve the data, convert it, and then check for missing values.

Objectives

After completing this lab, you will be able to:

- Write queries that include conversion functions.
- Write queries that use logical functions.
- Write queries that test for nullability.

Estimated Time: 40 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Writing Queries That Use Conversion Functions

Scenario

You have to prepare a couple of reports for the business users and the IT department.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement that Uses the CAST or CONVERT Function
3. Write a SELECT Statement to Filter Rows Based on Specific Date Information
4. Write a SELECT Statement to Convert the Phone Number Information to an Integer Value

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab08\Starter folder as Administrator.

▶ Task 2: Write a SELECT Statement that Uses the CAST or CONVERT Function

1. Open the project file D:\Labfiles\Lab08\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement against the Production.Products table to retrieve a calculated column named productdesc. The calculated column should be based on the productname and unitprice columns and look like this:

   Results: `The unit price for the Product HHYDP is 18.00 $.`

3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab08\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.

4. Did you use the CAST or the CONVERT function? Which one do you think is more appropriate to use?

▶ Task 3: Write a SELECT Statement to Filter Rows Based on Specific Date Information

1. The US marketing department has supplied you with a start date of 4/1/2007 (using US English form, read as April 1, 2007) and an end date of 11/30/2007 (using US English form, read as November 30, 2007). Write a SELECT statement against the Sales.Orders table to retrieve the orderid, orderdate, shippeddate, and shipregion columns. Filter the result to include only rows with the order date between the specified start date and end date and have more than 30 days between the shipped date and order date. Also check the shipregion column for missing values. If there is a missing value, then return the value 'No region'.
2. In this SELECT statement, you can use the CONVERT function with a style parameter or the PARSE function.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab08\Solution\53 - Lab Exercise 1 - Task 2 Result.txt.

▶ Task 4: Write a SELECT Statement to Convert the Phone Number Information to an Integer Value

1. The IT department would like to convert all the information about phone numbers in the Sales.Customers table to integer values. The IT staff indicated that all hyphens, parentheses, and spaces have to be removed before the conversion to an integer data type.
2. Write a SELECT statement to implement the requirement of the IT department. Replace all the specified characters in the phone column of the Sales.Customers table, and then convert the column from the nvarchar datatype to the int datatype. The T-SQL statement must not fail if there is a conversion error – it should return a NULL. (Hint: First try writing a T-SQL statement using the CONVERT function, and then compare it with the TRY_CONVERT function.) Use the alias phoneasint for this calculated column.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab08\Solution\54 - Lab Exercise 3 - Task 3 Result.txt.

Results: The unit price for the Product HHYDP is 18.00 $.

Exercise 2: Writing Queries That Use Logical Functions

Scenario

The sales department would like to have different reports regarding the segmentation of customers and specific order lines. You will add a new calculated column to show the target group for the segmentation.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Mark Specific Customers Based on their Country and Contact Title
2. Modify the T-SQL Statement to Mark Different Customers
3. Create Four Groups of Customers

▶ Task 1: Write a SELECT Statement to Mark Specific Customers Based on their Country and Contact Title

1. Open the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

2. Write a SELECT statement against the Sales.Customers table and retrieve the custid and contactname columns. Add a calculated column named segmentgroup, using a logical function IIF with the value "Target group" for customers that are from Mexico and have the value "Owner" in the contact title. Use the value "Other" for the rest of the customers.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab08\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Modify the T-SQL Statement to Mark Different Customers

1. Modify the T-SQL statement from task 1 to change the calculated column to show the value "Target group" for all customers without a missing value in the region attribute or with the value "Owner" in the contact title attribute.
2. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab08\Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Create Four Groups of Customers

1. Write a SELECT statement against the Sales.Customers table and retrieve the custid and contactname columns. Add a calculated column named segmentgroup using the logical function CHOOSE with four possible descriptions ("Group One", "Group Two", "Group Three", "Group Four"). Use the modulo operator on the column custid. (Use the expression custid % 4 + 1 to determine the target group.)
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab08\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

Results: After this exercise, you should know how to use the logical functions.

Exercise 3: Writing Queries That Test for Nullability

Scenario

The sales department would like to have additional segmentation of customers. Some columns that you should retrieve contain missing values, and you will have to change the NULL to some more meaningful information for the business users.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Customer Fax Information

2. Write a Filter for a Variable that Could Be a Null

3. Write a SELECT Statement to Return All the Customers that Do Not Have a Two-Character Abbreviation for the Region

▶ Task 1: Write a SELECT Statement to Retrieve the Customer Fax Information

1. Open the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the contactname and fax columns from the Sales.Customers table. If there is a missing value in the fax column, return the value 'No information'.
3. Write two solutions, one using the COALESCE function and the other using the ISNULL function.
4. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab08\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.
5. What is the difference between the ISNULL and COALESCE functions?

▶ Task 2: Write a Filter for a Variable that Could Be a Null

1. Update the provided T-SQL statement with a WHERE clause to filter the region column using the provided variable @region, which can have a value or a NULL. Test the solution using both provided variable declaration cases.

```
DECLARE @region AS NVARCHAR(30) = NULL;
SELECT
custid, region
FROM Sales.Customers;
GO
DECLARE @region AS NVARCHAR(30) = N'WA';
SELECT
custid, region
FROM Sales.Customers;
```

▶ Task 3: Write a SELECT Statement to Return All the Customers that Do Not Have a Two-Character Abbreviation for the Region

1. Write a SELECT statement to retrieve the contactname, city, and region columns from the Sales.Customers table. Return only rows that do not have two characters in the region column, including those with an inapplicable region (where the region is NULL).
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab08\Solution\73 - Lab Exercise 3 - Task 3 Result.txt. Notice the number of rows returned.

Results: After this exercise, you should have an understanding of how to test for nullability.

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways

- **Best Practice:** When possible, use standards-based functions such as CAST or COALESCE rather than SQL Server-specific functions like NULLIF or CONVERT.
- Consider the impact of functions in a WHERE clause on query performance.

Review Question(s)

Question: Which function should you use to convert from an int to a nchar(8)?

Question: Which function will return a NULL, rather than an error message, if it cannot convert a string to a date?

Question: What is the name for a function that returns a single value?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 9

Grouping and Aggregating Data

Contents:

Module Overview	**9-1**
Lesson 1: Using Aggregate Functions	**9-2**
Lesson 2: Using the GROUP BY Clause	**9-9**
Lesson 3: Filtering Groups with HAVING	**9-14**
Lab: Grouping and Aggregating Data	**9-17**
Module Review and Takeaways	**9-23**

Module Overview

In addition to row-at-a-time queries, you may need to summarize data in order to analyze it. Microsoft® SQL Server® provides a number of built-in functions that can aggregate, or summarize, information across multiple rows. In this module, you will learn how to use aggregate functions. You will also learn how to use the GROUP BY and HAVING clauses to break up the data into groups for summarizing and to filter the resulting groups.

Objectives

After completing this lesson, you will be able to:

- List the built-in aggregate functions provided by SQL Server.
- Write queries that use aggregate functions in a SELECT list to summarize all the rows in an input set.
- Describe the use of the DISTINCT option in aggregate functions.
- Write queries using aggregate functions that handle the presence of NULLs in source data.

Lesson 1
Using Aggregate Functions

In this lesson, you will learn how to use built-in functions to aggregate, or summarize, data in multiple rows. SQL Server provides functions such as SUM, MAX, and AVG to perform calculations that take multiple values and return a single result.

Lesson Objectives

After completing this lesson, you will be able to:

- List the built-in aggregate functions provided by SQL Server.
- Write queries that use aggregate functions in a SELECT list to summarize all the rows in an input set.
- Describe the use of the DISTINCT option in aggregate functions.
- Write queries using aggregate functions that handle the presence of NULLs in source data.

Working with Aggregate Functions

So far in this course, you have learned how to operate on a row at a time, using a WHERE clause to filter rows, adding computed columns to a SELECT list, and processing across columns but within each row.

You may also need to perform analysis across rows, such as counting rows that meet your criteria, or summarizing total sales for all orders. To accomplish this, you will use aggregate functions capable of operating on multiple rows at once.

- Aggregate functions:
  - Return a scalar value (with no column name)
  - Ignore NULLs except in COUNT(*)
  - Can be used in
    - SELECT, HAVING, and ORDER BY clauses
  - Frequently used with GROUP BY clause

```
SELECT    AVG(unitprice) AS avg_price,
          MIN(qty)AS min_qty,
          MAX(discount) AS max_discount
FROM Sales.OrderDetails;
```

```
avg_price min_qty max_discount
--------- ------- -----------
26.2185   1       0.250
```

There are many aggregate functions provided in SQL Server. In this course, you will learn about common functions such as SUM, MIN, MAX, AVG, and COUNT.

When working with aggregate functions, there are some considerations to keep in mind:

- Aggregate functions return a single (scalar) value and can be used in SELECT statements where a single expression is used, such as SELECT, HAVING, and ORDER BY clauses.
- Aggregate functions ignore NULLs, except when using COUNT(*). You will learn more about this later in the lesson.
- Aggregate functions in a SELECT list do not generate a column alias. You may wish to use the AS clause to provide one.
- Aggregate functions in a SELECT clause operate on all rows passed to the SELECT phase. If there is no GROUP BY clause, all rows will be summarized, as in the slide above. You will learn more about GROUP BY in the next lesson.

To extend beyond the built-in functions, SQL Server provides a mechanism for user-defined aggregate functions via the .NET Common Language Runtime (CLR).

For more information on other built-in aggregate functions, go to Books Online at:

MCT USE ONLY. STUDENT USE PROHIBITED

Aggregate Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402753

Built-In Aggregate Functions

SQL Server provides many built-in aggregate functions. Commonly used functions include:

Function Name	Syntax	Description
SUM	SUM(<expression>)	Totals all the non-NULL numeric values in a column.
AVG	AVG(<expression>)	Averages all the non-NULL numeric values in a column (sum/count).
MIN	MIN(<expression>)	Returns the smallest number, earliest date/time, or first-occurring string (according to sort rules in collation).
MAX	MAX(<expression>)	Returns the largest number, latest date/time, or last-occurring string (according to sort rules in collation).
COUNT or COUNT_BIG	COUNT(*) or COUNT(<expression>)	With (*), counts all rows, including those with NULL. When a column is specified as <expression>, returns count of non-NULL rows for the column. COUNT returns an int; COUNT_BIG returns a big_int.

This lesson will only cover common aggregate functions. For more information on other built-in aggregate functions, go to Books Online at:

Aggregate Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402753

To use a built-in aggregate in a SELECT clause, consider the following example in the TSQL sample database:

Aggregate Example

```
SELECT        AVG(unitprice) AS avg_price,
              MIN(qty)AS min_qty,
              MAX(discount) AS max_discount
FROM Sales.OrderDetails;
```

MCT USE ONLY STUDENT USE PROHIBITED

Note that the above example does not use a GROUP BY clause. Therefore, all rows from the Sales.OrderDetails table will be summarized by the aggregate formulas in the SELECT clause.

The results:

```
avg_price min_qty max_discount
--------- ------- ------------
26.2185   1       0.250
```

When using aggregates in a SELECT clause, all columns referenced in the SELECT list must be used as inputs for an aggregate function, or be referenced in a GROUP BY clause.

The following example will return an error:

Partial Aggregate Error

```
SELECT orderid, AVG(unitprice) AS avg_price, MIN(qty)AS min_qty, MAX(discount) AS
max_discount
FROM Sales.OrderDetails;
```

This returns:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Sales.OrderDetails.orderid' is invalid in the select list because it is not
contained in either an aggregate function or the GROUP BY clause.
```

Since our example is not using a GROUP BY clause, the query treats all rows as a single group. All columns, therefore, must be used as inputs to aggregate functions. Removing orderid from the previous example will prevent the error.

In addition to numeric data such as the price and quantities in the previous example, aggregate expressions can also summarize date, time, and character data. The following examples show the use of aggregates with dates and characters.

This query returns first and last company by name, using MIN and MAX:

Aggregating Character Data

```
SELECT MIN(companyname) AS first_customer, MAX(companyname) AS last_customer
FROM Sales.Customers;
```

Returns:

```
first_customer last_customer
-------------- --------------
Customer AHPOP Customer ZRNDE
```

This query returns the earliest and latest orders by order date, using MIN and MAX:

Aggregating Dates

```
SELECT MIN(orderdate)AS earliest,MAX(orderdate) AS latest
FROM Sales.Orders;
```

Returns:

```
earliest                latest
----------------------- -----------------------
2006-07-04 00:00:00.000 2008-05-06 00:00:00.000
```

Other functions may coexist with aggregate functions.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

For example, the YEAR scalar function is used in the following example to return only the year portion of the order date, before MIN and MAX are evaluated:

Aggregating with Functions

```
SELECT MIN(YEAR(orderdate))AS earliest, MAX(YEAR(orderdate)) AS latest
FROM Sales.Orders;
```

Returns:

```
earliest latest
-------- -------
2006     2008
```

Using DISTINCT with Aggregate Functions

Earlier in this course, you learned about the use of DISTINCT in a SELECT clause to remove duplicate rows. When used with an aggregate function, DISTINCT removes duplicate values from the input column before computing the summary value. This is useful when you wish to summarize unique occurrences of values, such as customers in the TSQL orders table.

- Use DISTINCT with aggregate functions to summarize only unique values
- DISTINCT aggregates eliminate duplicate values, not rows (unlike SELECT DISTINCT)
- Compare (with partial results):

```
SELECT empid, YEAR(orderdate) AS orderyear,
      COUNT(custid) AS all_custs,
      COUNT(DISTINCT custid) AS unique_custs
FROM Sales.Orders
GROUP BY empid, YEAR(orderdate);
```

empid	orderyear	all_custs	unique_custs
1	2006	26	22
1	2007	55	40
1	2008	42	32
2	2006	16	15

The following example returns customers who have placed orders, grouped by employee id and year:

Summarizing Distinct Values

```
SELECT empid, YEAR(orderdate) AS orderyear,
      COUNT(custid) AS all_custs,
      COUNT(DISTINCT custid) AS unique_custs
FROM Sales.Orders
GROUP BY empid, YEAR(orderdate);
```

Note that the above example uses a GROUP BY clause. GROUP BY will be covered in the next lesson. It is used here to provide a useful example for comparing DISTINCT and non-DISTINCT aggregate functions.

This returns, in part:

```
empid orderyear all_custs unique_custs
--------------- --------- ------------
1     2006      26        22
1     2007      55        40
1     2008      42        32
2     2006      16        15
2     2007      41        35
2     2008      39        34
3     2006      18        16
3     2007      71        46
3     2008      38        30
```

Note the difference in each row between the COUNT of custid (in column 3) and the DISTINCT COUNT in column 4. Column 3 simply returns all rows except those containing NULL. Column 4 excludes duplicate custids (repeat customers) and returns a count of unique customers, answering the question: "How many customers per employee?"

Question: Could you accomplish the same output with the use of SELECT DISTINCT?

Using Aggregate Functions with NULL

As you have learned in this course, it is important to be aware of the possible presence of NULLs in your data, and of how NULL interacts with T-SQL query components. This is also true with aggregate expressions. There are a few considerations to be aware of:

- Most aggregate functions ignore NULL
  - COUNT(<column>) ignores NULL
  - COUNT(*) counts all rows
- NULL may produce incorrect results (such as use of AVG)
- Use ISNULL or COALESCE to replace NULLs before aggregating

```
SELECT
    AVG(c2) AS AvgWithNULLs,
    AVG(COALESCE(c2,0)) AS AvgWithNULLReplace
FROM dbo.t2;
```

- With the exception of COUNT used with the (*) option, T-SQL aggregate functions ignore NULLs. This means, for example, that a SUM function will add only non-NULL values. NULLs do not evaluate to zero.
- The presence of NULLs in a column may lead to inaccurate computations for AVG, which will sum only populated rows and divide that sum by the number of non-NULL rows. There may be a difference in results between AVG(<column>) and (SUM(<column>)/COUNT(*)).

For example, given the following table named t1:

C1	C2
1	NULL
2	10
3	20
4	30
5	40
6	50

The following query illustrates the difference between how AVG handles NULL and how you might calculate an average with a SUM/COUNT(*) computed column:

Aggregating NULL Example

```
SELECT SUM(c2) AS sum_nonnulls,
      COUNT(*)AS count_all_rows,
      COUNT(c2)AS count_nonnulls,
      AVG(c2) AS [avg],
      (SUM(c2)/COUNT(*))AS arith_avg
FROM t1;
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The result:

```
sum_nonnulls count_all_rows count_nonnulls avg arith_avg
------------ -------------- -------------- --- ---------
150          6              5              30  25
```

If you need to summarize all rows, whether NULL or not, consider replacing the NULLs with another value that can be used by your aggregate function.

The following example replaces NULLs with 0 before calculating an average. The table named t2 contains the following rows:

```
c1          c2
----------- -----------
1           1
2           10
3           1
4           NULL
5           1
6           10
7           1
8           NULL
9           1
10          10
11          1
12          10
```

Compare the effect on the arithmetic mean with NULLs-ignored verses replaced with 0:

Replace NULLs with Zeros Example

```
SELECT AVG(c2) AS AvgWithNULLs, AVG(COALESCE(c2,0)) AS AvgWithNULLReplace
FROM dbo.t2;
```

This returns the following results, with a warning message:

```
AvgWithNULLs AvgWithNULLReplace
------------ ------------------
4            3
Warning: Null value is eliminated by an aggregate or other SET operation.
```

Note: This example cannot be executed against the sample database used in this course. A script to create the table is included in the upcoming demonstration.

Demonstration: Using Aggregate Functions

In this demonstration, you will see how to:

- Use built-in aggregate functions

Demonstration Steps

Use Built-in Aggregate Functions

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod09\Setup.cmd as an administrator.

3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod09\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

Question: You have the following query:

```
SELECT COUNT(*) AS RecordCount
FROM Sales.Products;
```

There are 250 records in the Products table. How many rows will be returned by this query?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Using the GROUP BY Clause

While aggregate functions are useful for analysis, you may wish to arrange your data into subsets before summarizing it. In this lesson, you will learn how to accomplish this using the GROUP BY clause.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that separate rows into groups using the GROUP BY clause.
- Describe the role of the GROUP BY clause in the logical order of operations for processing a SELECT statement.
- Write SELECT clauses that reflect the output of a GROUP BY clause.
- Use GROUP BY with aggregate functions.

Using the GROUP BY Clause

As you have learned, when your SELECT statement is processed, after the FROM clause and WHERE clause (if present) have been evaluated, a virtual table has been created. The contents of the virtual table are now available for further processing. The GROUP BY clause allows you to subdivide the results of the preceding query phases into groups of rows.

- GROUP BY creates groups for output rows, according to a unique combination of values specified in the GROUP BY clause

```
SELECT <select_list>
FROM <table_source>
WHERE <search_condition>
GROUP BY <group_by_list>;
```

- GROUP BY calculates a summary value for aggregate functions in subsequent phases

```
SELECT empid, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
```

- Detail rows are "lost" after GROUP BY clause is processed

To group rows, specify one or more elements in the GROUP BY clause:

GROUP BY Syntax

```
GROUP BY <value1> [, <value2>, ...]
```

GROUP BY creates groups and places rows into each group as determined by unique combinations of the elements specified in the clause.

For example, the following snippet of a query will result in a set of grouped rows, one per empid, in the Sales.Orders table:

GROUP BY Snippet

```
FROM Sales.Orders
GROUP BY empid;
```

Once the GROUP BY clause has been processed and rows have been associated with a group, subsequent phases of the query must aggregate any elements of the source rows that do not appear in the GROUP BY list. This will have an impact on how you write your SELECT and HAVING clauses.

To see into the results of the GROUP BY clause, you will need to add a SELECT clause.

This shows the original 830 source rows being grouped into nine groups based on the unique employee ID:

GROUP BY Example

```
SELECT empid, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
```

The result:

```
empid cnt
----- -----
1     123
2     96
3     127
4     156
5     42
6     67
7     72
8     104
9     43
(9 row(s) affected)
```

To learn more about GROUP BY, go to *GROUP BY (Transact SQL)* in Books Online at:

GROUP BY (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402754

GROUP BY and the Logical Order of Operations

A common obstacle to becoming comfortable with using GROUP BY in SELECT statements is understanding why the following type of error message occurs:

```
Msg 8120, Level 16, State 1, Line 2
Column <column_name> is invalid in the
select list because it is not contained
in either an aggregate function or the
GROUP BY clause.
```

Logical Order	Phase	Comments
5	SELECT	
1	FROM	
2	WHERE	
3	GROUP BY	Creates groups
4	HAVING	Operates on groups
6	ORDER BY	

- If a query uses GROUP BY, all subsequent phases operate on the groups, not source rows
- HAVING, SELECT, and ORDER BY must return a single value per group
- All columns in SELECT, HAVING, and ORDER BY must appear in GROUP BY clause or be inputs to aggregate expressions

A review of the logical order of operations during query processing will help clarify this issue.

If you recall from earlier in the course, the SELECT clause is not processed until after the FROM, WHERE, GROUP BY, and HAVING clauses are processed, if present. When discussing the use of GROUP BY, it is important to remember that not only does GROUP BY precede SELECT, but it also replaces the results of the FROM and WHERE clauses with its own results. The final outcome of the query will only return one row per qualifying group (if a HAVING clause is present). Therefore, any operations performed after GROUP BY, including SELECT, HAVING, and ORDER BY, are performed on the groups, not the original detail rows. Columns in the SELECT list, for example, must return a scalar value per group. This may include the column(s) being grouped on, or aggregate functions being performed on, each group.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The following query is permitted because each column in the SELECT list is either a column in the GROUP BY clause or an aggregate function operating on each group:

GROUP BY Example

```
SELECT empid, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
```

This returns:

```
empid count
----- -----
1     123
2     96
3     127
4     156
5     42
6     67
7     72
8     104
9     43
```

The following query will return an error since orderdate is not an input to GROUP BY, and its data has been "lost" following the FROM clause:

Missing GROUP BY Value

```
SELECT empid, orderdate, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
```

This returns:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Sales.Orders.orderdate' is invalid in the select list because it is not
contained in either an aggregate function or the GROUP BY clause.
```

If you did want to see orders per employee ID and per order date, add it to the GROUP BY clause, as follows:

Correct GROUP BY Example

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid, YEAR(orderdate)
ORDER BY empid, YEAR(orderdate);
```

This returns (in part):

```
empid orderyear count
----- --------- -----
1     2006      26
1     2007      55
1     2008      42
2     2006      16
2     2007      41
```

The net effect of this behavior is that you will not be able to combine a view of summary data with the detailed source date, using the T-SQL tools you have learned so far. You will learn some approaches to solving the problem later in this course.

For more information about troubleshooting GROUP BY errors, go to:

Troubleshooting GROUP BY Errors

http://go.microsoft.com/fwlink/?LinkID=402755

GROUP BY Workflow

Initially, the WHERE clause is processed followed by the GROUP BY. The slide shows the results of the WHERE clause, followed by the GROUP BY being performed on these results.

The source queries required to build the demonstration on the slide follow and are included with the demonstration file for this lesson:

Source Queries

```
SELECT orderid, empid, custid
FROM Sales.Orders;

SELECT orderid, empid, custid
FROM Sales.Orders
WHERE CUSTID <>3;

SELECT empid, COUNT(*)
FROM Sales.Orders
WHERE CUSTID <>3
GROUP BY empid;.
```

Using GROUP BY with Aggregate Functions

As you have heard, if you use a GROUP BY clause in a T-SQL query, all columns listed in the SELECT clause must either be used in the GROUP BY clause itself or be inputs to aggregate functions operating on each group.

You have seen the use of the COUNT function in conjunction with GROUP BY queries.

Other aggregate functions may also be used, as in the following example, which uses MAX to return the largest quantity ordered per product:

GROUP BY with Aggregate Example

```
SELECT productid, MAX(qty) AS largest_order
FROM Sales.OrderDetails
GROUP BY productid;
```

This returns (in part):

```
productid   largest_order
----------- -------------
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
23          70
46          60
69          65
29          80
75          120
```

Note: The qty column, used as an input to the MAX function, is not used in the GROUP BY clause. This illustrates that, even though the detail rows returned by the FROM...WHERE phase are lost to the GROUP BY phase, the source columns are still available for aggregation.

Demonstration: Using GROUP BY

In this demonstration, you will see how to:

- Use the GROUP BY clause.

Demonstration Steps

Use the GROUP BY Clause

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod09\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod09\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

   Question: You want to find out how many employees work in each department in your organization. You have the following query:

   ```
   SELECT d.DepartmentID, d.DepartmentName, COUNT(e.EmployeeID) AS
   EmployeeCount
   FROM HumanResources.Departments AS d
   INNER JOIN HumanResources.Employees AS e
   ON d.DepartmentID = e.DepartmentID
   GROUP BY d.DepartmentID;
   ```

   You run the query and receive an error. How can you modify the query in order for it to execute successfully?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Filtering Groups with HAVING

Once you have created groups with a GROUP BY clause, you may wish to further filter the results. The HAVING clause acts as a filter on groups, much like the WHERE clause acts as a filter on rows returned by the FROM clause. In this lesson, you will learn how to write a HAVING clause and understand the differences between HAVING and WHERE.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that use the HAVING clause to filter groups.
- Compare HAVING to WHERE.
- Choose the appropriate filter for a scenario: WHERE or HAVING.

Filtering Grouped Data Using the HAVING Clause

If a WHERE clause and a GROUP BY clause are present in a T-SQL SELECT statement, the HAVING clause is the fourth phase of logical query processing:

Logical Order	Phase	Comments
5	SELECT	
1	FROM	
2	WHERE	Operates on rows
3	GROUP BY	Creates groups
4	HAVING	Operates on groups
6	ORDER BY	

- HAVING clause provides a search condition that each group must satisfy
- HAVING clause is processed after GROUP BY

```
SELECT custid, COUNT(*) AS count_orders
FROM Sales.Orders
GROUP BY custid
HAVING COUNT(*) > 10;
```

A HAVING clause allows you to create a search condition, conceptually similar to the predicate of a WHERE clause, which will then test each group returned by the GROUP BY clause.

The following example from the TSQL database groups all orders by customer, then returns only those customers who have placed orders. No HAVING clause has been added so no filter is applied to the groups:

GROUP BY Without HAVING Clause

```
SELECT custid, COUNT(*) AS count_orders
FROM Sales.Orders
GROUP BY custid;
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY, STUDENT USE PROHIBITED

Returns the groups, with the following message:

```
(89 row(s) affected)
```

The following example adds a HAVING clause to the previous query. It groups all orders by customer, then returns only those who have placed 10 or more orders. Groups containing customers who placed fewer than 10 rows are discarded:

GROUP BY With HAVING Clause

```
SELECT custid, COUNT(*) AS count_orders
FROM Sales.Orders
GROUP BY custid
HAVING COUNT(*) >= 10;
```

Returns the groups with the following message:

```
(28 row(s) affected)
```

Note: Remember that HAVING is processed before the SELECT clause, so any column aliases created in a SELECT clause are not available to the HAVING clause.

Go to *HAVING (Transact-SQL)* in Books Online at:

HAVING (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402756

Compare HAVING to WHERE

While both HAVING and WHERE clauses filter data, it is important to remember that WHERE operates on rows returned by the FROM clause. If a GROUP BY...HAVING section exists in your query following a WHERE clause, the WHERE clause will filter rows before GROUP BY is processed, potentially limiting the groups that can be created.

A HAVING clause is processed after GROUP BY and only operates on groups, not detail rows. To summarize:

- A WHERE clause controls which rows are available to the next phase of the query.
- A HAVING clause controls which groups are available to the next phase of the query.

Note: WHERE and HAVING clauses are not mutually exclusive!

You will see a comparison of using WHERE and HAVING in the next demonstration.

- Using a COUNT(*) expression in HAVING clause is useful to solve common business problems:
- Show only customers that have placed more than one order:

```
SELECT c.custid, COUNT(*) AS cnt
FROM Sales.Customers AS c
    JOIN Sales.Orders AS o ON c.custid = o.custid
GROUP BY c.custid
HAVING COUNT(*) > 1;
```

- Show only products that appear on 10 or more orders:

```
SELECT p.productid, COUNT(*) AS cnt
FROM Production.Products AS p JOIN Sales.OrderDetails
    AS od ON p.productid = od.productid
GROUP BY p.productid
HAVING COUNT(*) >= 10;
```

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Filtering Groups with HAVING

In this demonstration, you will see how to:

- Filter grouped data using the HAVING clause.

Demonstration Steps

Filter Grouped Data Using the HAVING Clause

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod09\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod09\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

   Question: You are writing a query to count the number of orders placed for each product. You have the following query:

   ```
   SELECT p.ProductName, COUNT(*) AS OrderCount
   FROM Sales.Products AS p
   JOIN Sales.OrderItems AS o
   ON p.ProductID = o.ProductID
   GROUP BY p.ProductName;
   ```

   You want to change the query to return only products that cost more than $10. Should you add a HAVING clause or a WHERE clause?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Grouping and Aggregating Data

Scenario

You are a business analyst for Adventure Works who will be writing reports using corporate databases stored in SQL Server. You have been provided with a set of business requirements for data and you will write T-SQL queries to retrieve it from the databases. You will need to perform calculations upon groups of data and filter according to the results.

Objectives

After completing this lab, you will be able to:

- Write queries that use the GROUP BY clause.
- Write queries that use aggregate functions.
- Write queries that use distinct aggregate functions.
- Write queries that filter groups with the HAVING clause.

Estimated Time: 60 Minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Writing Queries That Use the GROUP BY Clause

Scenario

The sales department would like to create additional upsell opportunities from existing customers. The staff need to analyze different groups of customers and product categories, depending on several business rules. Based on these rules, you will write the SELECT statements to retrieve the needed rows from the Sales.Customers table.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement to Retrieve Different Groups of Customers
3. Add an Additional Column From the Sales.Customers Table
4. Write a SELECT Statement to Retrieve the Customers with Orders for Each Year
5. Write a SELECT Statement to Retrieve Groups of Product Categories Sold in a Specific Year

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab09\Starter folder as Administrator.

▶ Task 2: Write a SELECT Statement to Retrieve Different Groups of Customers

1. Open the project file D:\Labfiles\Lab09\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.

2. Write a SELECT statement that will return groups of customers who made a purchase. The SELECT clause should include the custid column from the Sales.Orders table and the contactname column from the Sales.Customers table. Group by both columns and filter only the orders from the sales employee whose empid equals five.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.

▶ Task 3: Add an Additional Column From the Sales.Customers Table

1. Copy the T-SQL statement in task 1 and modify it to include the city column from the Sales.Customers table in the SELECT clause.
2. Execute the query.
3. You will get an error. What is the error message? Why?
4. Correct the query so that it will execute properly.
5. Execute the query and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\53 - Lab Exercise 1 - Task 2 Result.txt.

▶ Task 4: Write a SELECT Statement to Retrieve the Customers with Orders for Each Year

1. Write a SELECT statement that will return groups of rows based on the custid column and a calculated column orderyear representing the order year based on the orderdate column from the Sales.Orders table. Filter the results to include only the orders from the sales employee whose empid equals five.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\54 - Lab Exercise 1 - Task 3 Result.txt.

▶ Task 5: Write a SELECT Statement to Retrieve Groups of Product Categories Sold in a Specific Year

1. Write a SELECT statement to retrieve groups of rows based on the categoryname column in the Production.Categories table. Filter the results to include only the product categories that were ordered in the year 2008.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\55 - Lab Exercise 1 - Task 4 Result.txt.

Results: After this exercise, you should be able to use the GROUP BY clause in the T-SQL statement.

Exercise 2: Writing Queries That Use Aggregate Functions

Scenario

The marketing department would like to launch a new campaign, so the staff need to gain a better insight into the existing customers' buying behavior. You will have to create different sales reports based on the total and average sales amount per year and per customer.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Total Sales Amount Per Order
2. Add Additional Columns
3. Write a SELECT Statement to Retrieve the Sales Amount Value Per Month

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

4. Write a SELECT Statement to List All Customers with the Total Sales Amount and Number of Order Lines Added

▶ Task 1: Write a SELECT Statement to Retrieve the Total Sales Amount Per Order

1. Open the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the orderid column from the Sales.Orders table and the total sales amount per orderid. (Hint: Multiply the qty and unitprice columns from the Sales.OrderDetails table.) Use the alias salesamount for the calculated column. Sort the result by the total sales amount in descending order.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Add Additional Columns

1. Copy the T-SQL statement in task 1 and modify it to include the total number of order lines for each order and the average order line sales amount value within the order. Use the aliases nooforderlines and avgsalesamountperorderline, respectively.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve the Sales Amount Value Per Month

1. Write a select statement to retrieve the total sales amount for each month. The SELECT clause should include a calculated column named yearmonthno (YYYYMM notation) based on the orderdate column in the Sales.Orders table and a total sales amount (multiply the qty and unitprice columns from the Sales.OrderDetails table). Order the result by the yearmonthno calculated column.
2. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab09\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

▶ Task 4: Write a SELECT Statement to List All Customers with the Total Sales Amount and Number of Order Lines Added

1. Write a select statement to retrieve all the customers (including those who did not place any orders) and their total sales amount, maximum sales amount per order line, and number of order lines.
2. The SELECT clause should include the custid and contactname columns from the Sales.Customers table and four calculated columns based on appropriate aggregate functions:
   a. totalsalesamount, representing the total sales amount per order
   b. maxsalesamountperorderline, representing the maximum sales amount per order line
   c. numberofrows, representing the number of rows (use * in the COUNT function)
   d. numberoforderlines, representing the number of order lines (use the orderid column in the COUNT function)
3. Order the result by the totalsalesamount column.
4. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\65 - Lab Exercise 2 - Task 4 Result.txt.
5. Notice that the custid 22 and 57 rows have a NULL in the columns with the SUM and MAX aggregate functions. What are their values in the COUNT columns? Why are they different?

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Writing Queries That Use Distinct Aggregate Functions

Scenario

The marketing department would like to have some additional reports that display the number of customers who made any order in the specific period of time and the number of customers based on the first letter in the contact name.

The main tasks for this exercise are as follows:

1. Modify a SELECT Statement to Retrieve the Number of Customers
2. Write a SELECT Statement to Analyze Segments of Customers
3. Write a SELECT Statement to Retrieve Additional Sales Statistics

▶ Task 1: Modify a SELECT Statement to Retrieve the Number of Customers

1. Open the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. A junior analyst prepared a T-SQL statement to retrieve the number of orders and the number of customers for each order year. Observe the provided T-SQL statement and execute it:

```
SELECT
YEAR(orderdate) AS orderyear,
COUNT(orderid) AS nooforders,
COUNT(custid) AS noofcustomers
FROM Sales.Orders
GROUP BY YEAR(orderdate);
```

3. Observe the results. Notice that the number of orders is the same as the number of customers. Why?
4. Correct the T-SQL statement to show the correct number of customers who placed an order for each year.
5. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Analyze Segments of Customers

1. Write a SELECT statement to retrieve the number of customers based on the first letter of the values in the contactname column from the Sales.Customers table. Add an additional column to show the total number of orders placed by each group of customers. Use the aliases firstletter, noofcustomers and nooforders. Order the result by the firstletter column.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve Additional Sales Statistics

1. Copy the T-SQL statement in exercise 1, task 5, and modify to include the following information about each product category – total sales amount, number of orders, and average sales amount per order. Use the aliases totalsalesamount, nooforders, and avgsalesamountperorder, respectively.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\74 - Lab Exercise 3 - Task 3 Result.txt.

Results: After this exercise, you should have an understanding of how to apply a DISTINCT aggregate function.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 4: Writing Queries That Filter Groups with the HAVING Clause

Scenario

The sales and marketing departments were satisfied with the reports you provided to analyze customers' behavior. Now they would like to have the results filtered, based on the total sales amount and number of orders. So, in the final exercise, you will learn how to filter the result based on aggregated functions and learn when to use the WHERE and HAVING clauses.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Top 10 Customers
2. Write a SELECT Statement to Retrieve Specific Orders
3. Apply Additional Filtering
4. Retrieve the Customers with More Than 25 Orders

▶ Task 1: Write a SELECT Statement to Retrieve the Top 10 Customers

1. Open the T-SQL script 81 - Lab Exercise 4.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the top 10 customers (by total sales amount) who spent more than $10,000 in terms of sales amount. Display the custid column from the Orders table and a calculated column that contains the total sales amount, based on the qty and unitprice columns from the Sales.OrderDetails table. Use the alias totalsalesamount for the calculated column.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Retrieve Specific Orders

1. Write a SELECT statement against the Sales.Orders and Sales.OrderDetails tables and display the empid column and a calculated column representing the total sales amount. Filter the results to group only the rows with an order year 2008.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\83 - Lab Exercise 4 - Task 2 Result.txt.

▶ Task 3: Apply Additional Filtering

1. Copy the T-SQL statement in task 2 and modify it to apply an additional filter to retrieve only the rows that have a sales amount higher than $10,000.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\84 - Lab Exercise 4 - Task 3_1 Result.txt.
3. Apply an additional filter to show only employees with empid equal number 3.
4. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\85 - Lab Exercise 4 - Task 3_2 Result.txt.
5. Did you apply the predicate logic in the WHERE clause or the HAVING clause? Which do you think is better? Why?

▶ Task 4: Retrieve the Customers with More Than 25 Orders

1. Write a SELECT statement to retrieve all customers who placed more than 25 orders and add information about the date of the last order and the total sales amount. Display the custid column from the Sales.Orders table and two calculated columns – lastorderdate based on the orderdate column and totalsalesamount based on the qty and unitprice columns in the Sales.OrderDetails table.

2. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab09\Solution\86 - Lab Exercise 4 - Task 4 Result.txt.

Results: After this exercise, you should have an understanding of how to use the HAVING clause.

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways

Review Question(s)

Question: What is the difference between the COUNT function and the COUNT_BIG function?

Question: Can a GROUP BY clause include more than one column?

Question: Can a WHERE clause and a HAVING clause in a query filter on the same column?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 10

Using Subqueries

Contents:

Module Overview	**10-1**
Lesson 1: Writing Self-Contained Subqueries	**10-2**
Lesson 2: Writing Correlated Subqueries	**10-7**
Lesson 3: Using the EXISTS Predicate with Subqueries	**10-11**
Lab: Using Subqueries	**10-14**
Module Review and Takeaways	**10-19**

Module Overview

At this point in the course, you have learned many aspects of the T-SQL SELECT statement, but each query you have written has been a single, self-contained statement. SQL Server also provides the ability to nest one query within another—in other words, to form subqueries. In a subquery, the results of the inner query (subquery) are returned to the outer query. This can provide a great deal of flexibility for your query logic. In this module, you will learn to write several types of subqueries.

Objectives

After completing this module, you will be able to:

- Describe the uses for queries that are nested within other queries.
- Write self-contained subqueries that return scalar or multi-valued results.
- Write correlated subqueries that return scalar or multi-valued results.
- Use the EXISTS predicate to efficiently check for the existence of rows in a subquery.

Lesson 1
Writing Self-Contained Subqueries

A subquery is a SELECT statement nested within another query. Being able to nest one query within another will enhance your ability to create effective queries in T-SQL. In this lesson, you will learn how to write self-contained queries, which are evaluated once, and provide their results to the outer query. You will learn how to write scalar subqueries, which return a single value, and multi-valued subqueries, which, as their name implies, can return a list of values to the outer query.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe where subqueries may be used in a SELECT statement.
- Write queries that use scalar subqueries in the WHERE clause of a SELECT statement.
- Write queries that use multi-valued subqueries in the WHERE clause of a SELECT statement.

Working with Subqueries

A subquery is a SELECT statement nested, or embedded, within another query. The nested query, which is the subquery, is the inner query. The query containing the nested query is the outer query.

- Subqueries are nested queries: queries within queries
- Results of inner query passed to outer query
  - Inner query acts like an expression from perspective of outer query
- Subqueries can be self-contained or correlated
  - Self-contained subqueries have no dependency on outer query
  - Correlated subqueries depend on values from outer query
- Subqueries can be scalar, multi-valued, or table-valued

The purpose of a subquery is to return results to the outer query. The form of the results will determine whether the subquery is a scalar or multi-valued subquery:

- Scalar subqueries, like scalar functions, return a single value. Outer queries need to be written to process a single result.
- Multi-valued subqueries return a result much like a single-column table. Outer queries need to be written to handle multiple possible results.

In addition to the choice between scalar and multi-valued subqueries, you may choose to write self-contained subqueries or others that are correlated with the outer query:

- Self-contained subqueries can be written as standalone queries, with no dependencies on the outer query. A self-contained subquery is processed once, when the outer query runs and passes its results to that outer query.
- Correlated subqueries reference one or more columns from the outer query and therefore depend on it. Correlated subqueries cannot be run separately from the outer query.

Note: You will learn about correlated subqueries later in this module.

Additional reading about subqueries can be found in Books Online at:

Subquery Fundamentals

http://go.microsoft.com/fwlink/?LinkID=402757

MCT USE ONLY. STUDENT USE PROHIBITED

Writing Scalar Subqueries

A scalar subquery is an inner SELECT statement within an outer query, written to return a single value. Scalar subqueries may be used anywhere in an outer T-SQL statement where a single-valued expression is permitted, such as in a SELECT clause, a WHERE clause, a HAVING clause, or even a FROM clause.

- Scalar subquery returns single value to outer query
- Can be used anywhere single-valued expression is used: SELECT, WHERE, and so on

```
SELECT orderid, productid, unitprice, qty
FROM Sales.OrderDetails
WHERE orderid =
    (SELECT MAX(orderid) AS lastorder
    FROM Sales.Orders);
```

- If inner query returns an empty set, result is converted to NULL
- Construction of outer query determines whether inner query must return a single value

- To write a scalar subquery, consider the following guidelines:
- To denote a query as a subquery, enclose it in parentheses.
- Multiple levels of subqueries are supported in SQL Server. In this lesson, we will only consider two-level queries (one inner query within one outer query), but up to 32 levels are supported.

If the subquery returns an empty set, the result of the subquery is converted and returned as a NULL. Ensure your outer query can gracefully handle a NULL in addition to other expected results.

To build the example query provided on the slide above, you may wish to start by writing and testing the inner query alone:

Inner Query

```
USE TSQL;
GO
SELECT MAX(orderid) AS lastorder
FROM Sales.Orders;
```

This returns:

```
lastorder
---------
11077
```

Then you will write the outer query, using the value returned by the inner query.

In this example, you will return details about the most recent order:

Outer and Inner Query

```
SELECT orderid, productid, unitprice, qty
FROM Sales.OrderDetails
WHERE orderid =
      (SELECT MAX(orderid) AS lastorder
      FROM Sales.Orders);
```

This returns (partial result):

```
orderid     productid   unitprice            qty
----------- ----------- -------------------- ------
11077       2           19.00                24
11077       3           10.00                4
11077       4           22.00                1
11077       6           25.00                1
```

MCT USE ONLY. STUDENT USE PROHIBITED

Test the logic of your subquery to ensure it will only return a single value. In the query above, since the outer query used an = operator in the predicate of the WHERE clause, an error would have been returned if the inner query returned more than one result. If the outer query is written to expect a single value, such as through the use of simple equality operators (=, <, >, <>, and so on), an error will be returned:

```
Msg 512, Level 16, State 1, Line 1
Subquery returned more than 1 value. This is not permitted when the subquery follows
=, !=, <, <= , >, >= or when the subquery is used as an expression.
```

In the case of the Sales.Orders table, orderid is known to be a unique column, enforced in the structure of the table by a PRIMARY KEY constraint.

Go to *PRIMARY KEY Constraints* in Books Online at:

PRIMARY KEY Constraints

http://go.microsoft.com/fwlink/?LinkID=402758

Writing Multi-Valued Subqueries

As its name suggests, a multi-valued subquery may return more than one result, in the form of a single-column set.

A multi-valued subquery is well suited to return results to the IN predicate, as in the following example:

- Multi-valued subquery returns multiple values as a single column set to the outer query
- Used with IN predicate
  - If any value in the subquery result matches IN predicate expression, the predicate returns TRUE

```
SELECT custid, orderid
FROM Sales.orders
WHERE custid IN (
    SELECT custid
    FROM Sales.Customers
    WHERE country = N'Mexico');
```

- May also be expressed as a JOIN (test both for performance)

Multi-Valued Subquery

```
SELECT custid, orderid
FROM Sales.orders
WHERE custid IN (
      SELECT custid
      FROM Sales.Customers
      WHERE country =N'Mexico');
```

In this example, if you were to execute only the inner query, you would return the following list of custids for customers in the country of Mexico:

```
custid
------
2
3
13
58
80
```

SQL Server will pass those results to the outer query, logically rewritten as follows:

Logical Structure of Outer Query

```
SELECT custid, orderid
FROM Sales.orders
WHERE custid IN (2,3,13,58,80);
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The outer query will continue to process the SELECT statement, with the following partial results:

```
custid orderid
------ -----------
2      10308
2      10625
3      10365
3      10507
3      10856
13     10259
58     10322
58     10354
```

As you continue to learn about writing T-SQL queries, you may find scenarios in which multi-valued subqueries are written as SELECT statements using JOINs.

For example, the previous subquery might be rewritten as follows, with the same results and comparable performance:

Subquery Rewritten as a Join

```
SELECT c.custid, o.orderid
FROM Sales.Customers AS c JOIN Sales.Orders AS o
      ON c.custid = o.custid
WHERE c.country = N'Mexico';
```

Note: In some cases, the database engine will interpret a subquery as a JOIN and execute it accordingly. As you learn more about SQL Server internals, such as execution plans, you may be able to see your queries interpreted this way. Go to Microsoft Course 20464D: *Developing Microsoft® SQL Server® Databases* for more information about execution plans and query performance.

Demonstration: Writing Self-Contained Subqueries

In this demonstration, you will see how to:

- Write a nested subquery.

Demonstration Steps

Write a Nested Subquery

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod10\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod10\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

Question: You are troubleshooting a query. The outer query contains an inner query in its WHERE clause. The first inner query also contains a second inner query in its WHERE clause. Both inner queries are self-contained. The complete query returns an error. How should you approach this task?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Writing Correlated Subqueries

Earlier in this module, you learned how to write self-contained subqueries, in which the inner query is independent of the outer query, executes once, and returns its results to the outer query. Microsoft SQL Server also supports correlated subqueries, in which the inner query receives input from the outer query and conceptually executes once per row in it. In this lesson, you will learn how to write correlated subqueries, as well as rewrite some types of correlated subqueries as JOINs for performance or logical efficiency.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how correlated subqueries are processed.
- Write queries that use correlated subqueries in a SELECT statement.
- Rewrite some correlated subqueries as JOINs.

Working with Correlated Subqueries

Like self-contained subqueries, correlated subqueries are SELECT statements nested within an outer query. They may also be written as scalar or multi-valued subqueries. They are typically used to pass a value from the outer query to the inner query, to be used as a parameter there.

- Correlated subqueries refer to elements of tables used in outer query
- Dependent on outer query, cannot be executed separately
  - Harder to test than self-contained subqueries
- Behaves as if inner query is executed once per outer row
- May return scalar value or multiple values

```
SELECT orderid, empid, orderdate
FROM Sales.Orders AS O1
WHERE orderdate =
        (SELECT MAX(orderdate)
         FROM Sales.Orders AS O2
         WHERE O2.empid = O1.empid)
ORDER BY empid, orderdate;
```

However, unlike self-contained subqueries, correlated subqueries depend on the outer query to pass values into the subquery as a parameter. This leads to some special considerations when planning their use:

- Correlated subqueries cannot be executed separately from the outer query. This complicates testing and debugging.
- Unlike self-contained subqueries which are processed once, correlated subqueries will run multiple times. Logically, the outer query runs first, and for each row returned, the inner query is processed.

The following example uses a correlated subquery to return the orders with the latest order date for each employee. The subquery accepts an input value from the outer query, uses the input in its WHERE clause, and returns a scalar result to the outer query. Line numbers have been added for use in the subsequent explanation. They do not indicate the order in which the steps are logically processed.

The following example uses a correlated subquery to return the orders with the latest order date for each employee:

Correlated Subquery Example

```
1.    SELECT orderid, empid, orderdate
2.    FROM Sales.Orders AS O1
3.    WHERE orderdate =
4.            (SELECT MAX(orderdate)
5.             FROM Sales.Orders AS O2
6.             WHERE O2.empid = O1.empid)
7.    ORDER BY empid, orderdate;
```

Let's examine this query and trace the role of each clause:

Line No.	Statement	Description
1	SELECT orderid, empid, orderdate	Columns returned by the outer query.
2	FROM Sales.Orders AS O1	Source table for the outer query. Note the alias.
3	WHERE orderdate =	Predicate used to evaluate the outer rows against the result of the inner query.
4	(SELECT MAX(orderdate)	Column returned by the inner query. Aggregate function returns a scalar value.
5	FROM Sales.Orders AS O2	Source table for the inner query. Note the alias.
6	WHERE O2.empid = O1.empid)	Correlation of empid from the outer query to empid from the inner query. This value will be supplied for each row in the outer query.
7	ORDER BY empid, orderdate;	Sorts the results of the outer query.

The query returns the following results. Note that some employees appear more than once, since they are associated with multiple orders on the latest orderdate:

```
orderid empid orderdate
----- ----- -----------------------
11077 1     2008-05-06 00:00:00.000
11073 2     2008-05-05 00:00:00.000
11070 2     2008-05-05 00:00:00.000
11063 3     2008-04-30 00:00:00.000
11076 4     2008-05-06 00:00:00.000
11043 5     2008-04-22 00:00:00.000
11045 6     2008-04-23 00:00:00.000
11074 7     2008-05-06 00:00:00.000
11075 8     2008-05-06 00:00:00.000
11058 9     2008-04-29 00:00:00.000
```

Question: Why can't a correlated subquery be executed separately from the outer query?

MCT USE ONLY. STUDENT USE PROHIBITED

Writing Correlated Subqueries

To write correlated subquery subqueries, consider the following guidelines:

- Write inner query to accept input value from outer query
- Write outer query to accept appropriate return result (scalar or multi-valued)
- Correlate queries by passing value from outer query to match argument in inner query

```
SELECT custid, orderid, orderdate
FROM Sales.Orders AS outerorders
WHERE orderdate =
        (SELECT MAX(orderdate)
         FROM Sales.Orders AS innerorders
         WHERE innerorders.custid = outerorders.custid)
ORDER BY custid;
```

- Write the outer query to accept the appropriate return result from the inner query. If the inner query will be scalar, you can use equality and comparison operators, such as =, <, >, and <> in the WHERE clause. If the inner query may return multiple values, use an IN predicate. Plan to handle NULL results.
- Identify the column from the outer query that will be passed to the correlated subquery. Declare an alias for the table that is the source of the column in the outer query.
- Identify the column from the inner table that will be compared to the column from the outer table. Create an alias for the source table, as you did for the outer query.
- Write the inner query to retrieve values from its source based on the input value from the outer query. For example, use the outer column in the WHERE clause of the inner query.

The correlation between the inner and outer queries occurs when the outer value is passed to the inner query for comparison. It's this correlation that gives the subquery its name.

Additional reading about correlated subqueries can be found in Books Online at:

Correlated Subqueries

http://go.microsoft.com/fwlink/?LinkID=402759

Demonstration: Writing Correlated Subqueries

In this demonstration, you will see how to:

- Write a correlated subquery.

Demonstration Steps

Write a Correlated Subquery

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod10\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod10\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

MCT USE ONLY. STUDENT USE PROHIBITED

Check Your Knowledge

Question	
Which of the following statements about correlated subqueries is correct?	
Select the correct answer.	
	To troubleshoot a correlated subquery, execute the inner query first on its own, before placing it into the outer query.
	In a correlated subquery, the inner query is run only once, regardless of the number of rows the outer query returns.
	In a correlated subquery, the inner query uses data returned by the outer query.
	In a correlated subquery, the inner query is executed first, the outer query second.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Using the EXISTS Predicate with Subqueries

In addition to retrieving values from a subquery, SQL Server provides a mechanism for checking whether any results would be returned from a query. The EXISTS predicate evaluates whether rows exist, but rather than return them, it returns TRUE or FALSE. This is a useful technique for validating data without incurring the overhead of retrieving and counting the results.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how the EXISTS predicate combines with a subquery to perform an existence test.
- Write queries that use EXISTS predicates in a WHERE clause to test for the existence of qualifying rows.

Working with EXISTS

When a subquery is invoked by an outer query using the EXISTS predicate, SQL Server handles the results of the subquery differently to what you have seen so far in this module. Rather than retrieve a scalar value or a multi-valued list from the subquery, EXISTS simply checks to see if there are any rows in the results.

- When a subquery is used with the keyword EXISTS, it functions as an existence test
  - True or false only - no rows passed back to outer query
- EXISTS evaluates to TRUE or FALSE (not UNKNOWN)
  - If any rows are returned by the subquery, EXISTS returns TRUE
  - If no rows are returned, EXISTS returns FALSE
- Syntax:

```
WHERE [NOT] EXISTS (subquery)
```

Conceptually, an EXISTS predicate is equivalent to retrieving the results, counting the rows returned, and comparing the count to zero. Compare the following queries, which will return details about employees who are associated with orders.

The first query uses COUNT in a subquery:

Using COUNT in a Subquery

```
SELECT empid, lastnamex
FROM HR.Employees AS e
WHERE (SELECT COUNT(*)
              FROM Sales.Orders AS O
              WHERE O.empid = e.empid)>0;
```

The second query, which returns the same results, uses EXISTS:

Using EXISTS in a Subquery

```
SELECT empid, lastname
FROM HR.Employees AS e
WHERE EXISTS( SELECT *
              FROM Sales.Orders AS O
              WHERE O.empid = e.empid);
```

In the first example, the subquery must count every occurrence of each empid found in the Sales.Orders table, and compare the count results to zero, simply to indicate that the employee has associated orders.

In the second query, EXISTS returns TRUE for an empid as soon as one has been found in the Sales.Orders table—a complete accounting of each occurrence is unnecessary.

Note: From the perspective of logical processing, the two query forms are equivalent. From a performance perspective, the database engine may treat the queries differently as it optimizes them for execution. Consider testing each one for your own usage.

Another useful application of EXISTS is negating it with NOT, as in the following example, which will return any customer who has never placed an order:

NOT EXISTS Example

```
SELECT custid, companyname
FROM Sales.Customers AS c
WHERE NOT EXISTS (
      SELECT *
      FROM Sales.Orders AS o
      WHERE c.custid=o.custid);
```

Once again, SQL Server will not have to return data about the related orders for customers who have placed orders. If a customer ID is found in the Sales.Orders table, NOT EXISTS evaluates to FALSE and the evaluation quickly completes.

Writing Queries Using EXISTS with Subqueries

To write queries that use EXISTS with subqueries, consider the following guidelines:

- The keyword EXISTS directly follows WHERE. No column name (or other expression) needs to precede it, unless NOT is also used.
- Within the subquery following EXISTS, the SELECT list only needs to contain (*). No rows are returned by the subquery, so no columns need to be specified.

Go to *Subqueries with EXISTS* in Books Online at:

Subqueries with EXISTS

http://go.microsoft.com/fwlink/?LinkId=242937

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Writing Subqueries Using EXISTS

In this demonstration, you will see how to:

- Write queries using EXISTS and NOT EXISTS

Demonstration Steps

Write Queries Using EXISTS and NOT EXISTS

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod10\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod10\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

   Question: The Human Resources database has recently been extended to record the skills possessed by employees. Employees have been added their skills to the database by using a web-based user interface. You want to find employees who have not added their skills yet. You have the following query:

```
SELECT e.EmployeeID, e.FirstName
FROM HumanResources.Employees AS e
WHERE NOT EXISTS (
   SELECT s.EmployeeID, s.SkillName, s.SkillCategory
   FROM HumanResources.Skills AS s
   WHERE e.EmployeeID = s.EmployeeID);
```

   How can you improve the query?

Lab: Using Subqueries

Scenario

You are a business analyst for Adventure Works, who will be writing reports using corporate databases stored in SQL Server. You have been handed a set of business requirements for data and you will write T-SQL queries to retrieve the specified data from the databases. Due to the complexity of some of the requests, you will need to embed subqueries into your queries to return results in a single query.

Objectives

After completing this lab, you will be able to:

- Write queries that use subqueries.
- Write queries that use scalar and multi-result set subqueries.
- Write queries that use correlated subqueries and the EXISTS predicate.

Estimated Time: 60 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **AdventureWorks\Student**

Password: **Pa$$w0rd**

Exercise 1: Writing Queries That Use Self-Contained Subqueries

Scenario

The sales department needs some advanced reports to analyze sales orders. You will write different SELECT statements that use self-contained subqueries.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement to Retrieve the Last Order Date
3. Write a SELECT Statement to Retrieve All Orders Placed on the Last Order Date
4. Observe the T-SQL Statement Provided by the IT Department
5. Write A SELECT Statement to Analyze Each Order's Sales as a Percentage of the Total Sales Amount

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.

   Run Setup.cmd in the D:\Labfiles\Lab10\Starter folder as Administrator.

▶ Task 2: Write a SELECT Statement to Retrieve the Last Order Date

1. Open the project file D:\Labfiles\Lab10\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to return the maximum order date from the table Sales.Orders.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab10\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Write a SELECT Statement to Retrieve All Orders Placed on the Last Order Date

1. Write a SELECT statement to return the orderid, orderdate, empid, and custid columns from the Sales.Orders table. Filter the results to include only orders where the date order equals the last order date. (Hint: Use the query in task 1 as a self-contained subquery.)
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab10\Solution\53 - Lab Exercise 1 - Task 2 Result.txt.

▶ Task 4: Observe the T-SQL Statement Provided by the IT Department

1. The IT department has written a T-SQL statement that retrieves the orders for all customers whose contact name starts with a letter I:

```
SELECT
orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE
custid =
(
SELECT custid
FROM Sales.Customers
WHERE contactname LIKE N'I%'
);
```

2. Execute the query and observe the result.
3. Modify the query to filter customers whose contact name starts with a letter B.
4. Execute the query. What happened? What is the error message? Why did the query fail?
5. Apply the needed changes to the T-SQL statement so that it will run without an error.
6. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab10\Solution\54 - Lab Exercise 1 - Task 3 Result.txt.

▶ Task 5: Write A SELECT Statement to Analyze Each Order's Sales as a Percentage of the Total Sales Amount

1. Write a SELECT statement to retrieve the orderid column from the Sales.Orders table and the following calculated columns:
   - totalsalesamount (based on the qty and unitprice columns in the Sales.OrderDetails table).
   - salespctoftotal (percentage of the total sales amount for each order divided by the total sales amount for all orders in a specific period).
2. Filter the results to include only orders placed in May 2008.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab10\Solution\55 - Lab Exercise 1 - Task 4 Result.txt.

Results: After this exercise, you should be able to use self-contained subqueries in T-SQL statements.

Exercise 2: Writing Queries That Use Scalar and Multi-Result Subqueries

Scenario

The marketing department would like to prepare materials for different groups of products and customers, based on historic sales information. You have to prepare different SELECT statements that use a subquery in the WHERE clause.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve Specific Products
2. Write a SELECT Statement to Retrieve Those Customers Without Orders
3. Add a Row and Rerun the Query That Retrieves Those Customers Without Orders

▶ Task 1: Write a SELECT Statement to Retrieve Specific Products

1. Open the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the productid and productname columns from the Production.Products table. Filter the results to include only products that were sold in high quantities (more than 100) for a specific order line.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab10\Solution\62 - Lab Exercise 2 -Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Retrieve Those Customers Without Orders

1. Write a SELECT statement to retrieve the custid and contactname columns from the Sales.Customers table. Filter the results to include only those customers who do not have any placed orders.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\63 - Lab Exercise 2 - Task 2 Result.txt. Remember the number of rows in the results.

▶ Task 3: Add a Row and Rerun the Query That Retrieves Those Customers Without Orders

1. The IT department has written a T-SQL statement that inserts an additional row in the Sales.Orders table. This row has a NULL in the custid column:

```
INSERT INTO Sales.Orders (
custid, empid, orderdate, requireddate, shippeddate, shipperid, freight,
shipname, shipaddress, shipcity, shipregion, shippostalcode, shipcountry)
VALUES
(NULL, 1, '20111231', '20111231', '20111231', 1, 0,
'ShipOne', 'ShipAddress', 'ShipCity', 'RA', '1000', 'USA')
```

2. Execute this query exactly as written inside a query window.
3. Copy the T-SQL statement you wrote in task 2 and execute it.
4. Observe the result. How many rows are in the result? Why?
5. Modify the T-SQL statement to retrieve the same number of rows as in task 2. (Hint: You have to remove the rows with an unknown value in the custid column.)
6. Execute the modified statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

Results: After this exercise, you should know how to use multi-result subqueries in T-SQL statements.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Writing Queries That Use Correlated Subqueries and an EXISTS Predicate

Scenario

The sales department would like to have some additional reports to display different analyses of existing customers. Because the requests are complex, you will need to use correlated subqueries.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Last Order Date for Each Customer

2. Write a SELECT Statement That Uses the EXISTS Predicate to Retrieve Those Customers Without Orders

3. Write a SELECT Statement to Retrieve Customers Who Bought Expensive Products

4. Write a SELECT Statement to Display the Total Sales Amount and the Running Total Sales Amount for Each Order Year

5. Clean the Sales.Customers Table

▶ Task 1: Write a SELECT Statement to Retrieve the Last Order Date for Each Customer

1. Open the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the custid and contactname columns from the Sales.Customers table. Add a calculated column named lastorderdate that contains the last order date from the Sales.Orders table for each customer. (Hint: You have to use a correlated subquery).
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement That Uses the EXISTS Predicate to Retrieve Those Customers Without Orders

1. Write a SELECT statement to retrieve all customers that do not have any orders in the Sales.Orders table, similar to the request in exercise 2, task 3. However, this time use the EXISTS predicate to filter the results to include only those customers without an order. Also, you do not need to explicitly check that the custid column in the Sales.Orders table is not NULL.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.
3. Why didn't you need to check for a NULL?

▶ Task 3: Write a SELECT Statement to Retrieve Customers Who Bought Expensive Products

1. Write a SELECT statement to retrieve the custid and contactname columns from the Sales.Customers table. Filter the results to include only customers that placed an order on or after April 1, 2008, and ordered a product with a price higher than $100.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\74 - Lab Exercise 3 - Task 3 Result.txt.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Write a SELECT Statement to Display the Total Sales Amount and the Running Total Sales Amount for Each Order Year

1. Running aggregates accumulate values over time. Write a SELECT statement to retrieve the following information for each year:
   - The order year.
   - The total sales amount.
   - The running total sales amount over the years. That is, for each year, return the sum of sales amount up to that year. So, for example, for the earliest year (2006), return the total sales amount, for the next year (2007), return the sum of the total sales amount for the previous year and 2007.
2. The SELECT statement should have three calculated columns:
   - orderyear, representing the order year. This column should be based on the orderyear column from the Sales.Orders table.
   - totalsales, representing the total sales amount for each year. This column should be based on the qty and unitprice columns from the Sales.OrderDetails table.
   - runsales, representing the running sales amount. This column should use a correlated subquery.
3. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\75 - Lab Exercise 3 - Task 4 Result.txt.

▶ Task 5: Clean the Sales.Customers Table

1. Delete the row added in exercise 2 using the provided SQL statement:

```
DELETE Sales.Orders
WHERE custid IS NULL;
```

2. Execute this query exactly as written inside a query window.

Results: After this exercise, you should have an understanding of how to use a correlated subquery in T-SQL statements.

Module Review and Takeaways

Review Question(s)

Question: Can a correlated subquery return a multi-valued set?

Question: What type of subquery may be rewritten as a JOIN?

Question: Which columns should appear in the SELECT list of a subquery following the EXISTS predicate?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 11

Using Table Expressions

Contents:

Module Overview	**11-1**
Lesson 1: Using Views	**11-2**
Lesson 2: Using Inline TVFs	**11-5**
Lesson 3: Using Derived Tables	**11-9**
Lesson 4: Using CTEs	**11-15**
Lab: Using Table Expressions	**11-18**
Module Review and Takeaways	**11-25**

Module Overview

Previously in this course, you learned about using subqueries as an expression that returned results to an outer calling query. Like subqueries, table expressions are query expressions, but table expressions extend this idea by allowing you to name them and work with the results as you would with data in any valid relational table. Microsoft® SQL Server® 2014 supports four types of table expressions: derived tables, common table expressions (CTEs), views, and inline table-valued functions (TVFs). In this module, you will learn to work with these forms of table expressions and how to use them to help create a modular approach to writing queries.

After completing this module, you will be able to:

- Create simple views and write queries against them.
- Create simple inline TVFs and write queries against them.
- Write queries that use derived tables.
- Write queries that use CTEs.

Note: Some of the examples used in this module have been adapted from samples published in *Microsoft SQL Server 2008 T-SQL Fundamentals* (Microsoft Press 2009).

Lesson 1
Using Views

The lifespans of some table expressions are limited to the query in which they are defined and invoked. Views and TVFs, however, can be persistently stored in a database and reused. A view is a table expression whose definition is stored in an SQL Server database. Like derived tables and CTEs, views are defined with SELECT statements. This provides not only the benefits of modularity and encapsulation possible with derived table and CTEs, but also adds reusability, as well as additional security beyond what is provided with query-scoped table expressions.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that return results from views.
- Create simple views.

Writing Queries That Return Results from Views

A view is a named table expression whose definition is stored as metadata in an SQL Server database. Views can be used as a source for queries in much the same way as tables themselves. However, views do not persistently store data; the definition of the view is unpacked at runtime and the source objects are queried.

- Views may be referenced in a SELECT statement just like a table
- Views are named table expressions with definitions stored in a database
- Like derived tables and CTEs, queries that use views can provide encapsulation and simplification
- From an administrative perspective, views can provide a security layer to a database

```
SELECT      <select_list>
FROM        <view_name>
ORDER BY    <sort_list>;
```

Note: In an indexed view, data is materialized in the view. Indexed views are beyond the scope of this course.

To write a query that uses a view as its data source, use the two-part view name wherever the table source would be used, such as in a FROM or a JOIN clause:

Querying a View Syntax

```
SELECT <select_list>
FROM <view_name>
ORDER BY <sort_list>;
```

Note that an ORDER BY clause is used in this sample syntax to emphasize the point that, as a table expression, there is no sort order included in the definition of a view. This will be discussed later in this lesson.

The following example uses a sample view whose definition is stored in the TSQL database. Note there is no way to determine that the FROM clause references a view and not a table:

Querying a View Example

```
SELECT custid, ordermonth, qty
FROM Sales.CustOrders;
```

MCT USE ONLY. STUDENT USE PROHIBITED

The partial results are indistinguishable from any other table-based query:

```
custid      ordermonth              qty
----------- ----------------------- -----------
7           2006-07-01 00:00:00.000 50
13          2006-07-01 00:00:00.000 11
14          2006-07-01 00:00:00.000 57
```

The apparent similarity between a table and a view provides an important benefit—an application can be written to use views instead of the underlying tables, shielding the application from changes to the tables. As long as the view continues to present the same structure to the calling application, the application will receive consistent results. Views can be considered an application programming interface (API) to a database for purposes of retrieving data.

Administrators can also use views as a security layer, granting users permissions to select from a view without providing permissions on the underlying source tables.

Additional Reading: For more information on database security, go to the Microsoft Course 20462D: *Administering a Microsoft SQL Server Database*.

Creating Simple Views

To use a view in your queries, it must be created by a database developer or administrator with appropriate permission in the database. While coverage of database security is beyond the scope of this course, you will have permission to create views in the lab database.

- Views are saved queries created in a database by administrators and developers
- Views are defined with a single SELECT statement
- ORDER BY is not permitted in a view definition without the use of TOP, OFFSET/FETCH, or FOR XML
  - To sort the output, use ORDER BY in the outer query
- View creation supports additional options beyond the scope of this class

```
CREATE VIEW HR.EmpPhoneList
AS
SELECT empid, lastname, firstname, phone
FROM HR.Employees;
```

To store a view definition, use the CREATE VIEW T-SQL statement to name and store a single SELECT statement. Note that the ORDER BY clause is not permitted in a view definition unless the view uses a TOP, OFFSET/FETCH, or FOR XML element.

This is the syntax of the CREATE VIEW statement:

CREATE VIEW Syntax

```
CREATE VIEW <schema_name.view_name> [<column_alias_list>]
[WITH <view_options>]
AS select_statement;
```

Note: This lesson covers the basics of creating views for the purposes of discussion about querying them only. For more information on views and view options, go to the Microsoft Course 20464D: *Developing Microsoft SQL Server Databases*.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The following example creates the view named Sales.CustOrders that exists in the TSQL sample database. Most of the code within the example makes up the definition of the SELECT statement itself:

CREATE VIEW Example

```
CREATE VIEW Sales.CustOrders
AS
SELECT
  O.custid,
  DATEADD(month, DATEDIFF(month, 0, O.orderdate), 0) AS ordermonth,
  SUM(OD.qty) AS qty
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
GROUP BY custid, DATEADD(month, DATEDIFF(month, 0, O.orderdate), 0);
```

You can query system metadata by querying system catalog views such as sys.views, which you will learn about in a later module.

To query a view, refer to it in the FROM clause of a SELECT statement, as you would refer to a table:

Querying a View Example

```
SELECT custid, ordermonth, qty
FROM Sales.CustOrders;
```

Demonstration: Using Views

In this demonstration, you will see how to:

- Create views.

Demonstration Steps

Create Views

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod11\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod11\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

.

Question: Your DBAs want to grant access to Sales users on the Customers table in the Sales database. However, they also need to prevent Sales users from reading values in the Customers.Relationship column. How can they setup this access?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Using Inline TVFs

An inline TVF is a form of table expression with several properties in common with views. Like a view, the definition of a TVF is stored as a persistent object in a database. Also like a view, an inline TVF encapsulates a single SELECT statement, returning a virtual table to the calling query. A primary distinction between a view and an inline TVF is that the latter can accept input parameters and refer to them in the embedded SELECT statement.

In this lesson, you will learn how to create basic inline TVFs and write queries that return results from them.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the structure and usage of inline TVFs.
- Use the CREATE FUNCTION statement to create simple inline TVFs.
- Write queries that return results from inline TVFs.

Writing Queries That Use Inline TVFs

Inline TVFs are named table expressions whose definitions are stored persistently in a database that can be queried in much the same way as a view. This enables reuse and centralized management of code in a way that is not possible for derived tables and CTEs as query-scoped table expressions.

- TVFs are named table expressions with definitions stored in a database
- TVFs return a virtual table to the calling query
- SQL Server provides two types of TVFs
  - Inline, based on a single SELECT statement
  - Multi-statement, which creates and loads a table variable
- Unlike views, TVFs support input parameters
- Inline TVFs may be thought of as parameterized views

Note: SQL Server supports several types of user-defined functions. In addition to inline TVFs, users can create scalar functions, multi-statement TVFs, and functions written in the .NET Common Language Runtime (CLR). For more information on these functions, go to the Microsoft course 20464D: *Developing Microsoft SQL Server 2014 Databases*.

One of the key distinctions between views and inline TVFs is that the latter can accept input parameters. Therefore, you may think of inline TVFs conceptually as parameterized views and choose to use them in place of views when flexibility of input is preferred.

Additional reading can be found in Books Online at:

CREATE FUNCTION (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402772

Creating Simple Inline TVFs

To use inline TVFs in your queries, they must be created by a database developer or administrator with appropriate permission in the database. While coverage of database security is beyond the scope of this course, you will have permission to create TVFs in the lab database.

- TVFs are created by administrators and developers
- Create and name function and optional parameters with CREATE FUNCTION
- Declare return type as TABLE
- Define inline SELECT statement following RETURN

```
CREATE FUNCTION Sales.fn_LineTotal (@orderid INT)
RETURNS TABLE
AS
RETURN
   SELECT orderid,
          CAST((qty * unitprice * (1 - discount)) AS
          DECIMAL(8, 2)) AS line_total
   FROM   Sales.OrderDetails
   WHERE  orderid = @orderid ;
```

To store an inline TVF view definition:

- Use the CREATE FUNCTION T-SQL statement to name and store a single SELECT statement with optional parameters.
- Use RETURNS TABLE to identify this function as a TVF.
- Enclose the SELECT statement inside parentheses following the RETURN keyword to make this an inline function.

Use the following syntax:

CREATE FUNCTION Syntax for Inline Table-Valued Functions

```
CREATE FUNCTION <schema.name>
(@<parameter_name> AS <data_type>, ...)
RETURNS TABLE
AS
RETURN (<SELECT_expression>);
```

The following example creates an inline TVF, which takes an input parameter to control how many rows are returned by the TOP operator:

Inline Table-Valued Function Example

```
CREATE FUNCTION Production.TopNProducts
(@t AS INT)
RETURNS TABLE
AS
RETURN
      (SELECT TOP (@t) productid, productname, unitprice
             FROM Production.Products
             ORDER BY unitprice DESC);
```

MCT USE ONLY. STUDENT USE PROHIBITED

Retrieving from Inline TVFs

After creating an inline TVF, you can invoke it by selecting from it, as you would a view. If there is an argument, you need to enclose it in parentheses. Multiple arguments need to be separated by commas.

Here is an example of how to query an inline TVF:

- SELECT from function
- Use two-part name
- Pass in parameters

```
SELECT orderid, line_total
FROM Sales.fn_LineTotal(10252) AS LT;
```

orderid	line_total
10252	2462.40
10252	47.50
10252	1088.00

Querying an Inline TVF

```
SELECT * FROM Production.TopNProducts(3)
```

The results:

```
productid productname   unitprice
--------- ------------- ---------
38        Product QDOMO 263.50
29        Product VJXYN 123.79
9         Product AOZBW 97.00
 (3 row(s) affected)
```

Note: Use of a two-part name is required when calling a user-defined function.

Demonstration: Inline TVFs

In this demonstration, you will see how to:

Create inline TVFs.

Demonstration Steps

Create Inline TVFs

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod11\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod11\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

MCT USE ONLY. STUDENT USE PROHIBITED

Check Your Knowledge

Question	
From the following statements, select the one that is true of TVFs but not true of Views.	
Select the correct answer.	
	Stored persistently in the database.
	Can accept input parameters.
	Can be referred to in a FROM clause, like a table.
	Does not stored data in the database but queries the database whenever it is called.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Using Derived Tables

In this lesson, you will learn how to write queries that create derived tables in the FROM clause of an outer query. You will also learn how to return results from the table expression defined in the derived table.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that create and retrieve results from derived tables.
- Describe how to provide aliases for column names in derived tables.
- Pass arguments to derived tables.
- Describe nesting and reuse behavior in derived tables.

Writing Queries with Derived Tables

Earlier in this course, you learned about subqueries, which are queries nested within other SELECT statements. Like subqueries, you create derived tables in the FROM clause of an outer SELECT statement. Unlike subqueries, you write derived tables using a named expression that is logically equivalent to a table and may be referenced as a table elsewhere in the outer query. Derived tables allow you to write T-SQL statements that are more modular, helping you break down complex queries into more manageable parts. Using derived tables in your queries can also provide workarounds for some of the restrictions imposed by the logical order of query processing, such as the use of column aliases.

To create a derived table, write the inner query between parentheses, followed by an AS clause and a name for the derived table:

Derived Table Syntax

```
SELECT <outer query column list>
FROM (SELECT <inner query column list>
      FROM <table source>) AS <derived table alias>
```

The following example uses a derived table to retrieve information about orders placed per year by distinct customers. The inner query builds a set of orders and places it into the derived table's derived year. The outer query operates on the derived table and summarizes the results.

The following example uses a derived table to retrieve information about orders placed by distinct customers per year:

Derived Table Example

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (SELECT YEAR(orderdate) AS orderyear, custid
      FROM Sales.Orders) AS derived_year
GROUP BY orderyear;
```

The results:

```
orderyear cust_count
--------- ----------
2006      67
2007      86
2008      81
 (3 row(s) affected)
```

When writing queries that use derived tables, consider the following:

- Derived tables are not stored in the database. Therefore, no special security privileges are required to write queries using derived tables, other than the rights to select from the source objects.
- A derived table is created at the time of execution of the outer query and goes out of scope when the outer query ends.

Derived tables do not necessarily have an impact on performance, compared to the same query expressed differently. When the query is processed, the statement is unpacked and evaluated against the underlying database objects.

Guidelines for Derived Tables

When writing queries that use derived tables, keep the following guidelines in mind:

- The nested SELECT statement that defines the derived table must have an alias assigned to it. The outer query will use the alias in its SELECT statement in much the same way you refer to aliased tables joined in a FROM clause.
- All columns referenced in the derived table's SELECT clause should be assigned aliases, a best practice that is not always required in T-SQL. Each alias must be unique within the expression. The column aliases may be declared inline with the columns or externally to the clause. You will see examples of this in the next topic.
- The SELECT statement that defines the derived table expression may not use an ORDER BY clause, unless it also includes a TOP operator, an OFFSET/FETCH clause, or a FOR XML clause. As a result, there is no sort order provided by the derived table. You sort the results in the outer query.
- The SELECT statement that defines the derived table may be written to accept arguments in the form of local variables. If the SELECT statement is embedded in a stored procedure, the arguments may be written as parameters for the procedure. You will see examples of this later in the module.
- Derived table expressions that are nested within an outer query can contain other derived table expressions. Nesting is permitted, but it is not recommended due to increased complexity and reduced readability.
- A derived table may not be referred to multiple times within an outer query. If you need to manipulate the same results, you will need to define the derived table expression every time, such as on each side of a JOIN operator.

Note: You will see examples of multiple usage of the same derived table expression in a query in the demonstration for this lesson.

MCT USE ONLY. STUDENT USE PROHIBITED

Using Aliases for Column Names in Derived Tables

To create aliases, you can use one of two methods – inline or external.

To define aliases inline or with the column specification, use the following syntax. Note that aliases are not required by T-SQL, but are a best practice:

- Column aliases may be defined inline:

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (
        SELECT YEAR(orderdate) AS orderyear, custid
        FROM Sales.Orders) AS derived_year
GROUP BY orderyear;
```

- Column aliases may be defined externally:

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (
        SELECT YEAR(orderdate), custid
        FROM Sales.Orders) AS
        derived_year(orderyear, custid)
GROUP BY orderyear;
```

Alias Syntax

```
SELECT <outer query column list>
FROM (SELECT <col1> AS <alias>, <col2> AS
<alias>...
      FROM <table_source>);
```

The following example declares aliases inline for the results of the YEAR function and the custid column:

Alias Example

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (SELECT YEAR(orderdate) AS orderyear, custid
      FROM Sales.Orders) AS derived_year
GROUP BY orderyear;
```

A partial result for the inner query displays the following:

```
orderyear   custid
----------- ------
2006        85
2006        79
2006        34
```

The inner results are passed to the outer query, which operates on the derived table's orderyear and custid columns:

```
orderyear   cust_count
----------- -----------
2006        67
2007        86
2008        81
```

To use externally declared aliases with derived tables, use the following syntax:

Declared Aliases with Derived Tables Syntax

```
SELECT <outer query column list>
FROM (SELECT <col1>, <col2>..
      FROM <table_source>) AS <derived_table_alias>(<col1_alias>, <col2_alias>);
```

The following example uses external alias definitions for orderyear and custid:

Declared Aliases with Derived Tables Example

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (SELECT YEAR(orderdate), custid
      FROM Sales.Orders) AS derived_year(orderyear, custid)
GROUP BY orderyear;
```

MCT USE ONLY. STUDENT USE PROHIBITED

Note: When using external aliases, if the inner query is executed separately, the aliases will not be returned to the outer query. For ease of testing and readability, it is recommended that you use inline rather than external aliases.

Passing Arguments to Derived Tables

Derived tables in SQL Server 2014 can accept arguments passed in from a calling routine, such as a T-SQL batch, function, or a stored procedure. Derived tables can be written with local variables serving as placeholders in their code. At runtime, the placeholders can be replaced with values supplied in the batch or with values passed as parameters to the stored procedure that invoked the query. This will allow your code to be reused more flexibly than rewriting the same query with different values each time.

- Derived tables may refer to arguments
- Arguments may be:
  - Variables declared in the same batch as the SELECT statement
  - Parameters passed into a table-valued function or stored procedure

```
DECLARE @emp_id INT = 9;
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (
        SELECT YEAR(orderdate) AS orderyear, custid
        FROM Sales.Orders
        WHERE empid=@emp_id
) AS derived_year
GROUP BY orderyear;
```

Note: The use of parameters in functions and stored procedures will be covered later in this course. This lesson focuses on writing table expressions that can accept arguments.

For example, the following batch declares a local variable (marked with the @ symbol) for the employee ID, and then uses the ability of SQL Server 2008 and later to assign a value to the variable in the same statement. The query accepts the @emp_id variable and uses it in the derived table expression:

Passing Arguments to Derived Tables

```
DECLARE @emp_id INT = 9; --declare and assign the variable
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (
      SELECT YEAR(orderdate) AS orderyear, custid
      FROM Sales.Orders
      WHERE empid=@emp_id --use the variable to pass a value to the derived table query
) AS derived_year
GROUP BY orderyear;
GO
```

The results:

```
orderyear   cust_count
----------- -----------
2006        5
2007        16
2008        16
 (3 row(s) affected)
```

Note: You will learn more about declaring variables, executing T-SQL code in batches, and working with stored procedures later in this class.

MCT USE ONLY STUDENT USE PROHIBITED

Nesting and Reusing Derived Tables

Since a derived table is itself a complete query expression, it is possible for that query to refer to a derived table expression. This creates a nesting scenario, which while possible, is not recommended for reasons of code maintenance and readability.

For example, the following query nests one derived table within another:

- Derived tables may be nested, though not recommended:

```
SELECT orderyear, cust_count
FROM (SELECT  orderyear,
        COUNT(DISTINCT custid) AS cust_count
        FROM ( SELECT YEAR(orderdate) AS orderyear
                ,custid
      FROM Sales.Orders) AS derived_table_1
        GROUP BY orderyear) AS derived_table_2
WHERE cust_count > 80;
```

- Derived tables may not be referred to multiple times in the same query
  - Each reference must be separately defined

Nested Derived Tables

```
SELECT orderyear, cust_count
FROM  (
      SELECT  orderyear, COUNT(DISTINCT
custid) AS cust_count
      FROM (
             SELECT YEAR(orderdate) AS orderyear ,custid
        FROM Sales.Orders) AS derived_table_1
      GROUP BY orderyear) AS derived_table_2
WHERE cust_count > 80;
```

Logically, the innermost query is processed first, returning these partial results as derived_table_1:

```
orderyear   custid
----------- -----------
2006        85
2006        79
2006        34
```

Next, the middle query runs, grouping and aggregating the results into derived_table_2:

```
orderyear   cust_count
----------- -----------
2006        67
2007        86
2008        81
```

Finally, the outer query runs, filtering the output:

```
orderyear   cust_count
----------- -----------
2007        86
2008        81
```

As you can see, while is possible to nest derived tables, it does add complexity.

While nesting derived tables is possible, references to the same derived table from multiple clauses of an outer query can be challenging. Since the table expression is defined in the FROM clause, subsequent phases of the query can see it, but it cannot be referenced elsewhere in the same FROM clause.

For example, a derived table defined in a FROM clause may be referenced in a WHERE clause, but not in a JOIN in the same FROM clause that defines it. The derived table must be defined separately, and multiple copies of the code maintained. For an alternative approach that allows reuse without maintaining separate copies of the derived table definition, see CTE discussion later in this module.

Question: How could you rewrite the previous example to eliminate one level of nesting?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Using Derived Tables

In this demonstration, you will see how to:

- Write queries that create derived tables.

Demonstration Steps

Write Queries that Create Derived Tables

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod11\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod11\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

   Question: You are troubleshooting the following query, which returns an error:

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (
   SELECT YEAR(orderdate) AS orderyear, custid
   FROM Sales.Orders
   WHERE empid = 354
      ORDER BY YEAR(orderdate)
) AS derived_year
GROUP BY orderyear;
```

   How can you resolve the error?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 4
Using CTEs

Another form of table expression provided by SQL Server 2014 is the CTE. Similar in some ways to derived tables, CTEs provide a mechanism for defining a subquery that may then be used elsewhere in a query. Unlike a derived table, a CTE is defined at the beginning of a query and may be referenced multiple times in the outer query.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the use of CTEs.
- Write queries that create CTEs and return results from the table expression.
- Describe how a CTE can be reused multiple times by the same outer query.

Writing Queries with CTEs

CTEs are named expressions defined in a query. Like subqueries and derived tables, CTEs provide a means to break down query problems into smaller, more modular units.

- CTEs are named table expressions defined in a query
- CTEs are similar to derived tables in scope and naming requirements
- Unlike derived tables, CTEs support multiple definitions, multiple references, and recursion

```
WITH <CTE_name>
AS (
    <CTE_definition>
        )
<outer query referencing CTE>;
```

When writing queries with CTEs, consider the following guidelines:

- Like derived tables, CTEs are limited in scope to the execution of the outer query. When the outer query ends, so does the CTE's lifetime.
- CTEs require a name for the table expression, as well as unique names for each of the columns referenced in the CTE's SELECT clause.
- CTEs may use inline or external aliases for columns.
- Unlike a derived table, a CTE may be referenced multiple times in the same query with one definition. Multiple CTEs may also be defined in the same WITH clause.
- CTEs support recursion, in which the expression is defined with a reference to itself. Recursive CTEs are beyond the scope of this course.

Additional reading on recursive CTEs may be found in Books Online at:

Recursive Queries Using Common Table Expressions

http://go.microsoft.com/fwlink/?LinkID=402773

Creating Queries with Common Table Expressions

To create a CTE, define it in a WITH clause, as in the following syntax:

CTE Syntax

```
WITH   <CTE_name>
AS     ( <CTE_definition> )
```

For example, the same query used to illustrate derived tables, when written to use a CTE, looks like this:

CTE Example

```
WITH CTE_year --name the CTE
AS -- define the subquery
(
      SELECT YEAR(orderdate) AS orderyear, custid
      FROM Sales.Orders
)
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM CTE_year --reference the CTE in the outer query
GROUP BY orderyear;
```

The results:

```
orderyear   cust_count
----------- -----------
2006        67
2007        86
2008        81
 (3 row(s) affected)
```

Demonstration: Using CTEs

In this demonstration, you will see how to:

- Write queries that create CTEs.

Demonstration Steps

Write Queries that Create CTEs

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod11\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod11\Demo folder.
3. In Solution Explorer, open the **41 – Demonstration D.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Check Your Knowledge

Question	
Which of the following features is required for a CTE query?	
Select the correct answer.	
	The query must have a WITH… AS clause.
	The query must include a GROUP BY clause.
	The query must include a CREATE FUNCTION statement.
	The query must include a nested derived query.

Lab: Using Table Expressions

Scenario

You are a business analyst for Adventure Works, who will be writing reports using corporate databases stored in SQL Server 2014. You have been provided with a set of business requirements for data and you will write T-SQL queries to retrieve the specified data from the databases. Because of advanced business requests, you will have to learn how to create and query different query expressions that represent a valid relational table.

Objectives

After completing this lab, you will be able to:

- Write queries that use views.
- Write queries that use derived tables.
- Write queries that use CTEs.
- Write queries that use Inline TVFs

Estimated Time: 90 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Writing Queries That Use Views

Scenario

In the last 10 modules, you had to prepare many different T-SQL statements to support different business requirements. Because some of them used a similar table and column structure, you would like to have them reusable. You will learn how to use one of two persistent table expressions—a view.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement to Retrieve All Products for a Specific Category
3. Write a SELECT Statement Against the Created View
4. Try to Use an ORDER BY Clause in the Created View
5. Add a Calculated Column to the View
6. Remove the Production.ProductsBeverages View

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab11\Starter folder as Administrator.

▶ Task 2: Write a SELECT Statement to Retrieve All Products for a Specific Category

1. In SQL Server Management Studio, open the project file D:\Labfiles\Lab11\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

2. Write a SELECT statement to return the productid, productname, supplierid, unitprice, and discontinued columns from the Production.Products table. Filter the results to include only products that belong to the category Beverages (categoryid equals 1).
3. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab11\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.
4. Modify the T-SQL code to include the following supplied T-SQL statement. Put this statement before the SELECT clause:

```
CREATE VIEW Production.ProductsBeverages AS
```

5. Execute the complete T-SQL statement. This will create an object view named ProductsBeverages under the Production schema.

▶ Task 3: Write a SELECT Statement Against the Created View

1. Write a SELECT statement to return the productid and productname columns from the Production.ProductsBeverages view. Filter the results to include only products where supplierid equals 1.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab11\Solution\53 - Lab Exercise 1 - Task 2 Result.txt.

▶ Task 4: Try to Use an ORDER BY Clause in the Created View

1. The IT department has written a T-SQL statement that adds an ORDER BY clause to the view created in task 1:

```
ALTER VIEW Production.ProductsBeverages AS
SELECT
productid, productname, supplierid, unitprice, discontinued
FROM Production.Products
WHERE categoryid = 1
ORDER BY productname;
```

2. Execute the provided code. What happened? What is the error message? Why did the query fail?
3. Modify the supplied T-SQL statement by including the TOP (100) PERCENT option. The query should look like this:

```
ALTER VIEW Production.ProductsBeverages AS
SELECT TOP(100) PERCENT
productid, productname, supplierid, unitprice, discontinued
FROM Production.Products
WHERE categoryid = 1
ORDER BY productname;
```

4. Execute the modified T-SQL statement. By applying the needed changes, you have altered the existing view. Notice that you are still using the ORDER BY clause.
5. If you write a query against the modified Production.ProductsBeverages view, is it guaranteed that the retrieved rows will be sorted by productname? Please explain.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 5: Add a Calculated Column to the View

1. The IT department has written a T-SQL statement that adds an additional calculated column to the view created in task 1:

```
ALTER VIEW Production.ProductsBeverages AS
SELECT
productid, productname, supplierid, unitprice, discontinued,
CASE WHEN unitprice > 100. THEN N'high' ELSE N'normal' END
FROM Production.Products
WHERE categoryid = 1;
```

2. Execute the provided query. What happened? What is the error message? Why did the query fail?
3. Apply the changes needed to get the T-SQL statement to execute properly.

▶ Task 6: Remove the Production.ProductsBeverages View

1. Remove the created view by executing the provided T-SQL statement:

```
IF OBJECT_ID(N'Production.ProductsBeverages', N'V') IS NOT NULL
DROP VIEW Production.ProductsBeverages;
```

2. Execute this code exactly as written inside a query window.

Results: After this exercise, you should know how to use a view in T-SQL statements.

Exercise 2: Writing Queries That Use Derived Tables

Scenario

The sales department would like to compare the sales amounts between the ordered year and the previous year to calculate the growth percentage. To prepare such a report, you will learn how to use derived tables inside T-SQL statements.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement Against a Derived Table
2. Write a SELECT Statement to Calculate the Total and Average Sales Amount
3. Write a SELECT Statement to Retrieve the Sales Growth Percentage

▶ Task 1: Write a SELECT Statement Against a Derived Table

1. Open the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement against a derived table and retrieve the productid and productname columns. Filter the results to include only the rows in which the pricetype column value is equal to high. Use the SELECT statement from exercise 1, task 4, as the inner query that defines the derived table. Do not forget to use an alias for the derived table. (You can use the alias p.)
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab11\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Write a SELECT Statement to Calculate the Total and Average Sales Amount

1. Write a SELECT statement to retrieve the custid column and two calculated columns: totalsalesamount, which returns the total sales amount per customer, and avgsalesamount, which returns the average sales amount of orders per customer. To correctly calculate the average sales amount of orders per customer, you should first calculate the total sales amount per order. You can do so by defining a derived table based on a query that joins the Sales.Orders and Sales.OrderDetails tables. You can use the custid and orderid columns from the Sales.Orders table and the qty and unitprice columns from the Sales.OrderDetails table.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve the Sales Growth Percentage

1. Write a SELECT statement to retrieve the following columns:
   - orderyear, representing the year of the order date.
   - curtotalsales, representing the total sales amount for the current order year.
   - prevtotalsales, representing the total sales amount for the previous order year.
   - percentgrowth, representing the percentage of sales growth in the current order year compared to the previous order year.
2. You will have to write a T-SQL statement using two derived tables. To get the order year and total sales columns for each SELECT statement, you can query an already existing view named Sales.OrderValues. The val column represents the sales amount.
3. Do not forget that the order year 2006 does not have a previous order year in the database, but it should still be retrieved by the query.
4. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

Results: After this exercise, you should be able to use derived tables in T-SQL statements.

Exercise 3: Writing Queries That Use CTEs

Scenario

The sales department needs an additional report showing the sales growth over the years for each customer. You could use your existing knowledge of derived tables and views, but instead you will practice how to use a CTE.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement that Uses a CTE

2. Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer

3. Write a SELECT Statement to Compare the Total Sales Amount for Each Customer Over the Previous Year

▶ Task 1: Write a SELECT Statement that Uses a CTE

1. Open the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement like the one in exercise 2, task 1, but use a CTE instead of a derived table. Use inline column aliasing in the CTE query and name the CTE ProductBeverages.

3. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer

1. Write a SELECT statement against Sales.OrderValues to retrieve each customer's ID and total sales amount for the year 2008. Define a CTE named c2008 based on this query using the external aliasing form to name the CTE columns custid and salesamt2008. Join the Sales.Customers table and the c2008 CTE, returning the custid and contactname columns from the Sales.Customers table and the salesamt2008 column from the c2008 CTE.
2. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Compare the Total Sales Amount for Each Customer Over the Previous Year

1. Write a SELECT statement to retrieve the custid and contactname columns from the Sales.Customers table. Also retrieve the following calculated columns:
   - salesamt2008, representing the total sales amount for the year 2008.
   - salesamt2007, representing the total sales amount for the year 2007.
   - percentgrowth, representing the percentage of sales growth between the year 2007 and 2008.
2. If percentgrowth is NULL, then display the value 0.
3. You can use the CTE from the previous task and add another one for the year 2007. Then join both of them with the Sales.Customers table. Order the result by the percentgrowth column.
4. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\74 - Lab Exercise 3 - Task 3 Result.txt.

Results: After this exercise, you should have an understanding of how to use a CTE in a T-SQL statement.

Exercise 4: Writing Queries That Use Inline TVFs

Scenario

You have learned how to write a SELECT statement against a view. However, since a view does not support parameters, you will now use an inline TVF to retrieve data as a relational table based on an input parameter.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer

2. Write a SELECT Statement Against the Inline TVF

3. Write a SELECT Statement to Retrieve the Top Three Products Based on the Total Sales Value for a Specific Customer

4. Using Inline TVFs, Write a SELECT Statement to Compare the Total Sales Amount for Each Customer Over the Previous Year

5. Remove the Created Inline TVFs

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

▶ Task 1: Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer

1. Open the T-SQL script 81 - Lab Exercise 4.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement against the Sales.OrderValues view and retrieve the custid and totalsalesamount columns as a total of the val column. Filter the results to include orders only for the year 2007.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.
4. Define an inline TVF using the following function header and add your previous query after the RETURN clause:

```
CREATE FUNCTION dbo.fnGetSalesByCustomer
(@orderyear AS INT) RETURNS TABLE
AS
RETURN
```

5. Modify the query by replacing the constant year value 2007 in the WHERE clause with the parameter @orderyear.
6. Highlight the complete code and execute it. This will create an inline TVF named dbo.fnGetSalesByCustomer.

▶ Task 2: Write a SELECT Statement Against the Inline TVF

1. Write a SELECT statement to retrieve the custid and totalsalesamount columns from the dbo.fnGetSalesByCustomer inline TVF. Use the value 2007 for the needed parameter.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\83 - Lab Exercise 4 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve the Top Three Products Based on the Total Sales Value for a Specific Customer

1. In this task, you will query the Production.Products and Sales.OrderDetails tables. Write a SELECT statement that retrieves the top three sold products based on the total sales value for the customer with ID 1. Return the productid and productname columns from the Production.Products table. Use the qty and unitprice columns from the Sales.OrderDetails table to compute each order line's value, and return the sum of all values per product, naming the resulting column totalsalesamount. Filter the results to include only the rows where the custid value is equal to 1.
2. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\84 - Lab Exercise 4 - Task 3_1 Result.txt.
3. Create an inline TVF based on the following function header, using the previous SELECT statement. Replace the constant custid value 1 in the query with the function's input parameter @custid:

```
CREATE FUNCTION dbo.fnGetTop3ProductsForCustomer
(@custid AS INT) RETURNS TABLE
AS
RETURN
```

4. Highlight the complete code and execute it. This will create an inline TVF named dbo.fnGetTop3ProductsForCustomer that accepts a parameter for the customer ID.
5. Test the created inline TVF by writing a SELECT statement against it and use the value 1 for the customer ID parameter. Retrieve the productid, productname, and totalsalesamount columns, and use the alias p for the inline TVF.

6. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\85 - Lab Exercise 4 - Task 3_2 Result.txt.

▶ Task 4: Using Inline TVFs, Write a SELECT Statement to Compare the Total Sales Amount for Each Customer Over the Previous Year

1. Write a SELECT statement to retrieve the same result as in exercise 3, task 3, but use the created TVF in task 2 (dbo.fnGetSalesByCustomer).
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\86 - Lab Exercise 4 - Task 4 Result.txt.

▶ Task 5: Remove the Created Inline TVFs

1. Remove the created inline TVFs by executing the provided T-SQL statement:

```
IF OBJECT_ID('dbo.fnGetSalesByCustomer') IS NOT NULL
DROP FUNCTION dbo.fnGetSalesByCustomer; IF
OBJECT_ID('dbo.fnGetTop3ProductsForCustomer') IS NOT NULL
DROP FUNCTION dbo.fnGetTop3ProductsForCustomer;
```

2. Execute this code exactly as written inside a query window.

Results: After this exercise, you should know how to use inline TVFs in T-SQL statements.

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways

Review Question(s)

Question: When would you use a CTE rather than a derived table for a query?

Question: Which table expressions allow variables to be passed in as parameters to the expression?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 12

Using Set Operators

Contents:

Module Overview	**12-1**
Lesson 1: Writing Queries with the UNION Operator	**12-2**
Lesson 2: Using EXCEPT and INTERSECT	**12-6**
Lesson 3: Using APPLY	**12-9**
Lab: Using Set Operators	**12-13**
Module Review and Takeaways	**12-18**

Module Overview

Microsoft® SQL Server® provides methods for performing operations using the sets that result from two or more different queries. In this module, you will learn how to use the set operators UNION, INTERSECT, and EXCEPT to compare rows between two input sets. You will also learn how to use forms of the APPLY operator to manipulate the rows in one table based on the output of a second table, which may be a derived table or a table-valued function (TVF).

Objectives

After completing this module, you will be able to:

- Write queries that combine data using the UNION operator.
- Write queries that compare sets using the INTERSECT and EXCEPT operators.
- Write queries that manipulate rows in a table by using APPLY, with the results of a derived table or function.

Lesson 1
Writing Queries with the UNION Operator

In this lesson, you will learn how to use the UNION operator to combine multiple input sets into a single result. UNION and UNION ALL provide a mechanism to add, in mathematical terms, one set to another, allowing you to stack result sets. UNION combines rows, compared to JOIN, which combines columns from different sources.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the conditions necessary to interact between input sets.
- Write queries that use UNION to combine input sets.
- Write queries that use UNION ALL to combine input sets.

Interactions Between Sets

SQL Server provides a number of set manipulation techniques, using a variety of set operators. To successfully work with these operators, it is important to understand some considerations for the input sets:

- The results of two input queries may be further manipulated
- Sets may be combined, compared, or operated against each other
- Both sets must have the same number of compatible columns
- ORDER BY not allowed in input queries, but may be used for result of set operation
- NULLs considered equal when comparing sets

```
<SELECT query_1>
<set_operator>
<SELECT query_2>
[ORDER BY <sort_list>]
```

- Each input set is the result of a query, which may include any SELECT statement components you have already learned about, with the exception of an ORDER BY clause.
- The input sets must have the same number of columns and the columns must have compatible data types. The column data types, if not initially compatible, must be made compatible through implicit conversion.
- A NULL in one set is treated as equal to a NULL in another, despite what you have learned about comparing NULLs earlier in this course.
- Each operator can be thought of as having two forms: DISTINCT and ALL. For example, UNION DISTINCT eliminates duplicate rows while combining two sets, and UNION ALL combines all rows, including duplicates. Not all set operators support both forms in SQL Server 2014.

Note: When working with set operators, it may be useful to remember that, in set theory, a set does not provide a sort order, and includes only distinct rows. If you need the results sorted, you will need to add an ORDER BY to the final results, as you may not use it inside the input queries.

MCT USE ONLY. STUDENT USE PROHIBITED

Using the UNION Operator

The UNION operator allows you to combine rows from one input set with rows from another into a resulting set. If a row appears in either of the input sets, it will be returned in the output.

For example, in the TSQL sample database, there are 29 rows in the Production.Suppliers table and 91 rows in the Sales.Customers table.

Simply combining all rows from each set should yield 29 + 91, or 120 rows. Yet because of duplicates, UNION returns 93 rows in this example:

UNION Example

```
SELECT country, city
FROM Production.Suppliers
UNION
SELECT country, city
FROM Sales.Customers;
```

A partial result:

```
country   city
--------- ---------------
Argentina Buenos Aires
Australia Melbourne
...
USA       Walla Walla
Venezuela Barquisimeto
Venezuela Caracas
Venezuela I. de Margarita
Venezuela San Cristóbal
 (93 row(s) affected)
End of RLO
```

Note: Remember that there is no sort order guaranteed by set operators. Although the results might appear to be sorted, this is a by-product of the filtering performed and is not assured. If you require sorted output, add an ORDER BY clause at the end of the second query.

As previously mentioned, set operators can conceptually be thought of in two forms: DISTINCT and ALL. SQL Server does not implement an explicit UNION DISTINCT, though it does implement UNION ALL. ANSI SQL standards do specify both as explicit forms (UNION DISTINCT and UNION ALL). In T-SQL, the use of DISTINCT is not supported but is the implicit default. UNION combines all rows from each input set, and then filters out duplicates.

From a performance standpoint, the use of UNION will include a filter operation, whether or not there are duplicate rows. If you need to combine sets and already know that there are no duplicates, consider using UNION ALL to save the overhead of the distinct filter.

Note: You will learn about UNION ALL in the next lesson.

Go to *UNION (Transact-SQL)* in Books Online at:

MCT USE ONLY. STUDENT USE PROHIBITED

UNION (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402774

Using the UNION ALL Operator

If you wish to return all rows from both input sets, or know there will be no duplicates to filter out, you can use the UNION ALL operator.

The following example continues from the previous topic, and combines all supplier locations with all customer locations to yield all rows from each input set:

- UNION ALL returns a result set with all rows from both sets
- To avoid performance penalty, use UNION ALL even if you know there are no duplicates

```
-- all rows from both queries will be returned
SELECT country, region, city FROM HR.Employees
UNION ALL
SELECT country, region, city FROM Sales.Customers;
```

UNION ALL Example

```
SELECT country, city
FROM Production.Suppliers
UNION ALL
SELECT Country, City
FROM Sales.Customers;
```

This time, the result does include all 91 + 29 rows, partially displayed below:

```
country city
------- ---------------
UK      London
USA     New Orleans
...
Finland Helsinki
Poland  Warszawa
 (120 row(s) affected)
```

Since UNION ALL does not perform any filtering of duplicates, it can be used in place of UNION in cases where you know there will be no duplicate input rows and wish to improve performance, compared to using UNION.

Demonstration: Using UNION and UNION ALL

In this demonstration, you will see how to:

- Use UNION and UNION ALL.

Demonstration Steps

Use UNION and UNION ALL

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod12\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod12\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.

MCT USE ONLY STUDENT USE PROHIBITED

6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

   Question: Explain the way T-SQL treats NULL in WHERE clauses and contrast this with the way T-SQL treats NULL in UNION queries.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Using EXCEPT and INTERSECT

While UNION and UNION ALL combine all rows from input sets, you may need to return either only those rows in one set but not in the other – or only rows that are present in both sets. For these purposes, the EXCEPT and INTERSECT operators may be useful to your queries. You will learn how to use EXCEPT and INTERSECT in this lesson.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that use the EXCEPT operator to return only rows in one set but not another.
- Write queries that use the INTERSECT operator to return only rows that are present in both sets.

Using the INTERSECT Operator

The T-SQL INTERSECT operator, added in SQL Server 2005, returns only distinct rows that appear in both input sets.

Note: While UNION supports both conceptual forms DISTINCT and ALL, INTERSECT currently only provides an implicit DISTINCT option. No duplicate rows will be returned by the operation.

The following example uses INTERSECT to return geographical information in common between customers and suppliers. Remember that there are 91 rows in the Customers table and 29 in the Suppliers table:

INTERSECT Example

```
SELECT country, city
FROM Production.Suppliers
INTERSECT
SELECT country, city
FROM Sales.Customers;
```

The results:

```
country  city
-------- ---------
Germany  Berlin
UK       London
Canada   Montréal
France   Paris
Brazil   Sao Paulo
 (5 row(s) affected)
```

Go to *EXCEPT and INTERSECT (Transact-SQL)* in Books Online at:

EXCEPT and INTERSECT (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402775

Using the EXCEPT Operator

The T-SQL EXCEPT operator, added in SQL Server 2005, returns only distinct rows that appear in one set and not the other. Specifically, EXCEPT returns rows from the input set listed first in the query. As with queries that use a LEFT OUTER JOIN, the order in which the inputs are listed is important.

Note: While UNION supports both conceptual forms DISTINCT and ALL, EXCEPT currently only provides an implicit DISTINCT option. No duplicate rows will be returned by the operation.

The following example uses EXCEPT to return geographical information that is not common between the customers and suppliers. Remember that there are 91 rows in the Customers table and 29 in the Suppliers table. Initially, the query is executed with the Suppliers table listed first:

EXCEPT Example

```
SELECT country, city
FROM Production.Suppliers
EXCEPT
SELECT country, city
FROM Sales.Customers;
```

This returns 24 rows, partially displayed here:

```
country    city
---------- -------------
Australia  Melbourne
Australia  Sydney
Canada     Ste-Hyacinthe
Denmark    Lyngby
Finland    Lappeenranta
France     Annecy
France     Montceau
 (24 row(s) affected)
```

The results are different when the order of the input sets is reversed:

Input Set Order Reversed Example

```
SELECT country, city
FROM Sales.Customers
EXCEPT
SELECT country, city
FROM Production.Suppliers;
```

This returns 64 rows. When using EXCEPT, plan the order of the input queries carefully.

MCT USE ONLY STUDENT USE PROHIBITED

Demonstration: Using EXCEPT and INTERSECT

In this demonstration, you will see how to:

- Use INTERSECT and EXCEPT.

Demonstration Steps

Use INTERSECT and EXCEPT

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod12\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod12\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

   Question: You have a table of employees and a table of customers. You want to find out which countries have at least one customer and at least one employee. Should you use a UNION, EXCEPT, or INTERSECT operator?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Using APPLY

As an alternative to combining or comparing rows from two sets, SQL Server provides a mechanism to apply a table expression from one set on each row in the other set. In this lesson, you will learn how to use the APPLY operator to process rows in one set using rows in another.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the use of the APPLY operator to manipulate sets.
- Write queries using the CROSS APPLY operator.
- Write queries using the OUTER APPLY operator.

Using the APPLY Operator

SQL Server provides the SQL Server-specific APPLY operator to enable queries that evaluate rows in one input set against the expression that defines the second input set. Strictly speaking, APPLY is a table operator, not a set operator. You will use APPLY in a FROM clause, like a JOIN, rather than as a set operator that operates on two compatible result sets of queries.

- APPLY is a table operator used in the FROM clause
- Includes CROSS APPLY and OUTER APPLY
- Operates on two input tables, left and right
- Right table is often a derived table or a table-valued function

```
SELECT <column_list>
FROM       <left_table> AS <alias>
CROSS/OUTER APPLY
   <derived_table_expression or inline_TVF>
   AS <alias>
```

Conceptually, the APPLY operator is similar to a correlated subquery in that it applies a correlated table expression to each row from a table. However, APPLY differs from correlated subqueries by returning a table-valued result rather than a scalar or multi-valued result. For example, the table expression could be a TVF, and you can pass elements from the left row as input parameters to the TVF. The TVF will be logically processed once for each row in the left table. This will be demonstrated later.

Note: When describing input tables used with APPLY, the terms "left" and "right" are used in the same way as they are with the JOIN operator, based on the order they are listed in the FROM clause.

To use APPLY, you will supply two input sets within a single FROM clause. Unlike the set operators you have learned about earlier in this lesson, with APPLY, the second, or right, input may be a TVF that will be logically processed once per row found in the other input. The TVF will typically take values found in columns from the left input and use them as parameters within the function.

Additionally, APPLY supports two different forms: CROSS APPLY and OUTER APPLY, which you will learn about in this lesson.

The general syntax for APPLY lists the derived table or TVF second, or on the right, of the other input table:

APPLY Syntax

```
SELECT <column_list>
FROM <left_table> AS <alias>
[CROSS]|[OUTER] APPLY
      <derived_table_expression or inline_TVF> AS <alias>
```

You will learn how CROSS APPLY and OUTER APPLY work in the next topics.

Go to Using APPLY in the "Remarks" section of *FROM (Transact-SQL)* in Books Online at:

FROM (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402776

Also see Using APPLY at:

Using APPLY

http://go.microsoft.com/fwlink/?LinkID=402777

Using the CROSS APPLY Operator

CROSS APPLY correlates the rows in the left table expression against the derived table or table-valued expression in the right input. This enables you to write queries that go beyond comparing and combining data. You can perform more flexible manipulations, such as TOP N per group, which will be demonstrated later in this module.

- CROSS APPLY applies the right table expression to each row in left table
  - Conceptually similar to CROSS JOIN between two tables but can correlate data between sources

```
SELECT S.supplierid, s.companyname, P.productid,
       P.productname, P.unitprice
FROM Production.Suppliers AS S
CROSS APPLY
       dbo.fn_TopProductsByShipper(S.supplierid) AS P
```

The usage of CROSS in CROSS APPLY may be suggestive of a CROSS JOIN, in which all rows in the left and right tables are joined. However, if there is no result output from the right expression, the current row on the left will not be returned. In this regard, a CROSS APPLY may be conceptually closer to an INNER JOIN.

To use a CROSS APPLY, list the TVF or derived table second in the query, supplying parameters if needed. The TVF or derived table will be logically processed once for each row in the left table.

The following example uses the supplierid column from the left input table as an input parameter to a TVF named dbo.fn_TopProductsByShipper. If there are rows in the Suppliers table with no corresponding products, the rows will not be displayed:

CROSS APPLY Example

```
SELECT S.supplierid, s.companyname, P.productid, P.productname, P.unitprice
FROM Production.Suppliers AS S
CROSS APPLY dbo.fn_TopProductsByShipper(S.supplierid) AS P;
```

MCT USE ONLY. STUDENT USE PROHIBITED

Partial results appear as follows:

```
supplierid  companyname    productid   productname   unitprice
----------- -------------- ----------- ------------- ---------
1           Supplier SWRXU 2           Product RECZE 19.00
1           Supplier SWRXU 1           Product HHYDP 18.00
1           Supplier SWRXU 3           Product IMEHJ 10.00
2           Supplier VHQZD 4           Product KSBRM 22.00
2           Supplier VHQZD 5           Product EPEIM 21.35
2           Supplier VHQZD 65          Product XYWBZ 21.05
3           Supplier STUAZ 8           Product WVJFP 40.00
3           Supplier STUAZ 7           Product HMLNI 30.00
3      Supplier STUAZ 6     Product VAIIV 25.00
```

Note: Code to create this example function, as well as to test it, is provided in the demonstration script for this lesson.

Using the OUTER APPLY Operator

OUTER APPLY correlates the rows in the left table expression against the derived table or table-valued expression in the right input. Like CROSS APPLY, the table expression on the right will be processed once for each row in the left input table. However, where CROSS APPLY did not return rows where the right expression had an empty result, OUTER APPLY will add rows for the left table where NULL was returned on the right. The usage of OUTER in OUTER APPLY is conceptually similar to a LEFT OUTER JOIN, in which all rows in the left table are joined to matching rows in the right table and NULLs are added.

- Conceptually similar to LEFT OUTER JOIN between two tables
- OUTER APPLY adds a step to the processing used by CROSS APPLY:
  1. OUTER APPLY applies the right table expression to each row in left table
  2. OUTER APPLY adds rows for those with NULL in columns for right table

```
SELECT S.supplierid, s.companyname,
       P.productid, P.unitprice
FROM Production.Suppliers AS S
OUTER APPLY
       dbo.fn_TopProductsByShipper(S.supplierid) AS P
```

The following example uses the custid column from the left input table as an input parameter to a derived table that accepts the custid and uses it to find corresponding orders. If there are rows in the Customers table with no corresponding orders, the rows will be displayed with NULL for order attributes:

OUTER APPLY Example

```
SELECT C.custid, TopOrders.orderid, TopOrders.orderdate
FROM Sales.Customers AS C
OUTER APPLY
      (SELECT TOP (3) orderid, orderdate
       FROM Sales.Orders AS O
       WHERE O.custid = C.custid
      ORDER BY orderdate DESC, orderid DESC) AS TopOrders;
```

MCT USE ONLY. STUDENT USE PROHIBITED

Partial results, including rows with NULLs, appear as follows:

```
custid      orderid     orderdate
----------- ----------- -----------
1           11011       2008
1           10952       2008
1           10835       2008
2           10926       2008
2           10759       2007
2           10625       2007
22          NULL        NULL
57          NULL        NULL
58          11073       2008
58          10995       2008
58          10502       2007
(265 row(s) affected)
```

Demonstration: Using CROSS APPLY and OUTER APPLY

In this demonstration, you will see how to:

- Use APPLY.

Demonstration Steps

Use APPLY

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod12\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod12\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

   Question: You want to correlate the results from a query against the HumanResources.Employees table with results from a TVF named MostSkilledEmployees(). The TVF returns only 20 or fewer employees but you want to examine results for all employees. Should you use a CROSS APPLY or an OUTER APPLY?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Using Set Operators

Scenario

You are a business analyst for Adventure Works, who will be writing reports using corporate databases stored in SQL Server 2014. You have been provided with a set of business requirements for data and you will write T-SQL queries to retrieve the specified data from the databases. Because of the complex business requirements, you will need to prepare combined results from multiple queries using set operators.

Objectives

After completing this lab, you will be able to:

- Write queries that use the UNION and UNION ALL operators.
- Write queries that use the CROSS APPLY and OUTER APPLY operators.
- Write queries that use the EXCEPT and INTERSECT operators.

Estimated Time: 60 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Writing Queries That Use UNION Set Operators and UNION ALL Multi-Set Operators

Scenario

The marketing department needs some additional information regarding segmentation of products and customers. It would like to have a report, based on multiple queries, which is presented as one result. You will write different SELECT statements, and then merge them together into one result using the UNION operator.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment

2. Write a SELECT Statement to Retrieve Specific Products

3. Write a SELECT Statement to Retrieve All Products with a Total Sales Amount of More than $50,000

4. Merge the Results from Task 1 and Task 2

5. Write a SELECT Statement to Retrieve the Top 10 Customers by Sales Amount for January 2008 and February 2008

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab12\Starter folder as Administrator.

▶ Task 2: Write a SELECT Statement to Retrieve Specific Products

1. In SQL Server Management Studio, open the project file D:\Labfiles\Lab12\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.

2. Write a SELECT statement to return the productid and productname columns from the Production.Products table. Filter the results to include only products that have a categoryid value 4.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab12\ Solution\52 - Lab Exercise 1 - Task 1 Result.txt. Remember the number of rows in the results.

▶ Task 3: Write a SELECT Statement to Retrieve All Products with a Total Sales Amount of More than $50,000

1. Write a SELECT statement to return the productid and productname columns from the Production.Products table. Filter the results to include only products that have a total sales amount of more than $50,000. For the total sales amount, you will need to query the Sales.OrderDetails table and aggregate all order line values (qty * unitprice) for each product.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab12\ Solution\53 - Lab Exercise 1 - Task 2 Result.txt. Remember the number of rows in the results.

▶ Task 4: Merge the Results from Task 1 and Task 2

1. Write a SELECT statement that uses the UNION operator to retrieve the productid and productname columns from the T-SQL statements in task 1 and task 2.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab12\ Solution\54 - Lab Exercise 1 - Task 3_1 Result.txt.
3. What is the total number of rows in the results? If you compare this number with an aggregate value of the number of rows from tasks 1 and 2, is there any difference?
4. Copy the T-SQL statement and modify it to use the UNION ALL operator.
5. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab12\ Solution\55 - Lab Exercise 1 - Task 3_2 Result.txt.
6. What is the total number of rows in the result? What is the difference between the UNION and UNION ALL operators?

▶ Task 5: Write a SELECT Statement to Retrieve the Top 10 Customers by Sales Amount for January 2008 and February 2008

1. Write a SELECT statement to retrieve the custid and contactname columns from the Sales.Customers table. Display the top 10 customers by sales amount for January 2008 and display the top 10 customers by sales amount for February 2008 (Hint: Write two SELECT statements, each joining Sales.Customers and Sales.OrderValues and use the appropriate set operator).
2. Execute the T-SQL code and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab12\ Solution\56 - Lab Exercise 1 - Task 4 Result.txt.

Results: After this exercise, you should know how to use the UNION and UNION ALL set operators in T-SQL statements.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Writing Queries That Use the CROSS APPLY and OUTER APPLY Operators

Scenario

The sales department needs a more advanced analysis of buying behavior. The sales staff want to find out the top three products, based on sales revenue, for each customer. Therefore, you will need to use the APPLY operator.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement That Uses the CROSS APPLY Operator to Retrieve the Last Two Orders for Each Product

2. Write a SELECT Statement That Uses the CROSS APPLY Operator to Retrieve the Top Three Products, Based on Sales Revenue, for Each Customer

3. Use the OUTER APPLY Operator

4. Analyze the OUTER APPLY Operator

5. Remove the Created Inline TVF

▶ Task 1: Write a SELECT Statement That Uses the CROSS APPLY Operator to Retrieve the Last Two Orders for Each Product

1. Open the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the productid and productname columns from the Production.Products table. In addition, for each product, retrieve the last two rows from the Sales.OrderDetails table based on orderid number.
3. Use the CROSS APPLY operator and a correlated subquery. Order the result by the column productid.
4. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab12\ Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement That Uses the CROSS APPLY Operator to Retrieve the Top Three Products, Based on Sales Revenue, for Each Customer

1. Execute the provided T-SQL code to create the inline TVF fnGetTop3ProductsForCustomer, as you did in the previous module:

```
IF OBJECT_ID('dbo.fnGetTop3ProductsForCustomer') IS NOT NULL
DROP FUNCTION dbo.fnGetTop3ProductsForCustomer;
GO
CREATE FUNCTION dbo.fnGetTop3ProductsForCustomer
(@custid AS INT) RETURNS TABLE
AS
RETURN
SELECT TOP(3)
d.productid,
p.productname,
SUM(d.qty * d.unitprice) AS totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
INNER JOIN Production.Products AS p ON p.productid = d.productid
WHERE custid = @custid
GROUP BY d.productid, p.productname
ORDER BY totalsalesamount DESC;
```

2. Write a SELECT statement to retrieve the custid and contactname columns from the Sales.Customers table. Use the CROSS APPLY operator with the dbo.fnGetTop3ProductsForCustomer function to retrieve productid, productname, and totalsalesamount columns for each customer.

3. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab12\ Solution\63 - Lab Exercise 2 - Task 2 Result.txt. Remember the number of rows in the results.

▶ Task 3: Use the OUTER APPLY Operator

1. Copy the T-SQL statement from the previous task and modify it by replacing the CROSS APPLY operator with the OUTER APPLY operator.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab12\ Solution\64 - Lab Exercise 2 - Task 3 Result.txt. Notice that you achieved more rows than in the previous task.

▶ Task 4: Analyze the OUTER APPLY Operator

1. Copy the T-SQL statement from the previous task and modify it by filtering the results to show only customers without products. (Hint: In a WHERE clause, look for any column returned by the inline TVF that is NULL.)
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab12\ Solution\65 - Lab Exercise 2 - Task 4 Result.txt.
3. What is the difference between the CROSS APPLY and OUTER APPLY operators?

▶ Task 5: Remove the Created Inline TVF

1. Remove the created inline TVF by executing the provided T-SQL code:

```
IF OBJECT_ID('dbo.fnGetTop3ProductsForCustomer') IS NOT NULL
DROP FUNCTION dbo.fnGetTop3ProductsForCustomer;
```

2. Execute this code exactly as written inside a query window.

Results: After this exercise, you should be able to use the CROSS APPLY and OUTER APPLY operators in your T-SQL statements.

Exercise 3: Writing Queries That Use the EXCEPT and INTERSECT Operators

Scenario

The marketing department was satisfied with the results from exercise 1, but the staff now need to see specific rows from one result set that are not present in the other result set. You will have to write different queries using the EXCEPT and INTERSECT operators.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Return All Customers Who Bought More than 20 Distinct Products

2. Write a SELECT Statement to Retrieve All Customers from the USA, Except Those Who Bought More than 20 Distinct Products

3. Write a SELECT Statement to Retrieve Customers Who Spent More than $10,000

4. Write a SELECT Statement That Uses the EXCEPT and INTERSECT Operators

5. Change the Operator Precedence

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 1: Write a SELECT Statement to Return All Customers Who Bought More than 20 Distinct Products

1. Open the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the custid column from the Sales.Orders table. Filter the results to include only customers who bought more than 20 different products (based on the productid column from the Sales.OrderDetails table).
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab12\ Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Retrieve All Customers from the USA, Except Those Who Bought More than 20 Distinct Products

1. Write a SELECT statement to retrieve the custid column from the Sales.Orders table. Filter the results to include only customers from the country USA and exclude all customers from the previous (task 1) result. (Hint: Use the EXCEPT operator and the previous query).
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab12\ Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve Customers Who Spent More than $10,000

1. Write a SELECT statement to retrieve the custid column from the Sales.Orders table. Filter only customers who have a total sales value greater than $10,000. Calculate the sales value using the qty and unitprice columns from the Sales.OrderDetails table.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab12\ Solution\74 - Lab Exercise 3 - Task 3 Result.txt.

▶ Task 4: Write a SELECT Statement That Uses the EXCEPT and INTERSECT Operators

1. Copy the T-SQL statement from task 2. Add the INTERSECT operator at the end of the statement. After the INTERSECT operator, add the T-SQL statement from task 3.
2. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab12\ Solution\75 - Lab Exercise 3 - Task 4 Result.txt. Notice the total number of rows in the results.
3. In business terms, can you explain which customers are part of the result?

▶ Task 5: Change the Operator Precedence

1. Copy the T-SQL statement from the previous task and add parentheses around the first two SELECT statements (from the beginning until the INTERSECT operator).
2. Execute the T-SQL statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab12\ Solution\76 - Lab Exercise 3 - Task 5 Result.txt. Notice the total number of rows in the results.
3. Are the results different to the results from task 4? Please explain why.
4. What is the precedence among the set operators?

Results: After this exercise, you should have an understanding of how to use the EXCEPT and INTERSECT operators in T-SQL statements.

Module Review and Takeaways

Review Question(s)

Question: Which set operator would you use to combine sets if you knew there were no duplicates and wanted better performance?

Question: Which APPLY form will not return rows from the left table if the result of the right table expression was empty?

Question: What is the difference between APPLY and JOIN?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 13

Using Window Ranking, Offset, and Aggregate Functions

Contents:

Module Overview	**13-1**
Lesson 1: Creating Windows with OVER	**13-2**
Lesson 2: Exploring Window Functions	**13-8**
Lab: Using Window Ranking, Offset, and Aggregate Functions	**13-16**
Module Review and Takeaways	**13-20**

Module Overview

Microsoft® SQL Server® implements support for SQL windowing operations, which means you can define a set of rows and apply several different functions against those rows. Once you have learned how to work with windows and window functions, you may find that some types of queries which appeared to require complex manipulations of data (for example, self-joins, temporary tables, and other constructs) aren't needed to write your reports.

Objectives

After completing this module, you will be able to:

- Describe the benefits of using window functions.
- Restrict window functions to rows defined in an OVER clause, including partitions and frames.
- Write queries that use window functions to operate on a window of rows and return ranking, aggregation, and offset comparison results.

Lesson 1
Creating Windows with OVER

SQL Server provides a number of window functions, which perform calculations such as ranking, aggregations, and offset comparisons between rows. To use these functions, you will need to write queries that define windows, or sets, of rows. You will use the OVER clause and its related elements to define the sets for the window functions.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the T-SQL components used to define windows, and the relationships between them.
- Write queries that use the OVER clause, with partitioning, ordering, and framing to define windows.

SQL Windowing

SQL Server provides windows as a method for applying functions to sets of rows. There are many applications of this technique that solve common problems in writing T-SQL queries. For example, using windows allows the easy generation of row numbers in a result set and the calculation of running totals. Windows also provide an efficient way to compare values in one row with values in another without needing to join a table to itself using an inequality operator.

There are several core elements of writing queries that use windows:

1. Windows allow you to specify an order to rows that will be passed to a window function, without affecting the final order of the query output.
2. Windows include a partitioning feature, which enables you to specify that you want to restrict a function only to rows that have the same value as the current row.
3. Windows provide a framing option. It allows you to specify a further subset of rows within a window partition by setting upper and lower boundaries for the window frame, which presents rows to the window function.

The following example uses an aggregate window function to calculate a running total. This illustrates the use of these elements:

Running Total Example

```
SELECT Category, Qty, Orderyear,
      SUM(Qty) OVER (PARTITION BY Category  ORDER BY Orderyear
              ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS RunningQty
FROM Sales.CategoryQtyYear;
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The partial results:

```
Category        Qty   Orderyear  RunningQty
--------------- ----- ---------- -----------
Beverages       1842  2006       1842
Beverages       3996  2007       5838
Beverages       3694  2008       9532
Condiments      962   2006       962
Condiments      2895  2007       3857
Condiments      1441  2008       5298
Confections     1357  2006       1357
Confections     4137  2007       5494
Confections     2412  2008       7906
Dairy Products  2086  2006       2086
Dairy Products  4374  2007       6460
Dairy Products  2689  2008       9149
```

During the next few topics of this lesson, you will learn how to use these query elements.

Windowing Components

In order to use windows and window functions in T-SQL, you will always use one of the subclauses that create and manipulate windows—the OVER subclause. Additionally, you may need to create partitions with the PARTITION BY option, and even further restrict which rows are applied to a function with framing options. Therefore, understanding the relationship between these components is vital.

The general relationship can be expressed as a sequence, with one element further manipulating the rows output by the previous element:

1. The OVER clause determines the result set that will be used by the window function. An OVER clause with no partition defined is unrestricted. It returns all rows to the function.
2. A PARTITION BY clause, if present, restricts the results to those rows with the same value in the partitioned columns as the current row. For example, PARTITION BY custid restricts the window to rows with the same custid as the current row. PARTITION BY builds on the OVER clause and cannot be used without OVER. (An OVER clause without a window partition clause is considered one partition).
3. A ROW or RANGE clause creates a window frame within the window partition, which allows you to set a starting and ending boundary around the rows being operated on. A frame requires an ORDER BY subclause within the OVER clause.

The following example, also seen in the previous topic, aggregates the Qty column against a window in the OVER clause defined by partitioning on the category column, sorting on the orderyear and framing by a boundary at the first row and a boundary at the current row. This creates a "moving window," where each row is aggregated with other rows of the same category value, from the oldest row by orderyear, to the current row:

MCT USE ONLY. STUDENT USE PROHIBITED

Windowing Example

```
SELECT Category, Qty, Orderyear,
      SUM(Qty) OVER (PARTITION BY category ORDER BY Orderyear
              ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS RunningQty
FROM Sales.CategoryQtyYear;
```

The details of each component will be covered in future topics.

Note: A single query can use multiple window functions, each with its own OVER clause. Each clause determines its own partitioning, ordering, and framing.

Using OVER

The OVER clause defines the window, or set, of rows that will be operated on by a window function, which we will look at in the next lesson. The OVER clause includes partitioning, ordering, and framing, where each is applicable.

- OVER defines a window, or set, of rows to be used by a window function, including any ordering
- With a specified window partition clause, the OVER clause restricts the set of rows to those with the same values in the partitioning elements
- By itself, OVER() is unrestricted and includes all rows
- Multiple OVER clauses can be used in a single query, each with its own partitioning and ordering, if needed

```
OVER ( [ <PARTITION BY clause> ]
       [ <ORDER BY clause> ]
       [ <ROWS or RANGE clause> ]
       )
```

Used alone, the OVER clause does not restrict the result set passed to the window function. Used with a PARTITION BY subclause, OVER restricts the set to those rows with the same values in the partitioning elements.

The following example shows the use of OVER without an explicit window partition to define an unrestricted window that will be used by the ROW_NUMBER function. All rows will be numbered, using an ORDER BY clause, which is required by ROW_NUMBER. The row numbers will be displayed in a new column named Running:

OVER Example

```
SELECT Category, Qty, Orderyear,
      ROW_NUMBER() OVER (ORDER BY Qty DESC) AS Running
FROM Sales.CategoryQtyYear
ORDER BY Running;
```

The partial result, further ordered by the Running column for display purposes:

```
Category        Qty         Orderyear   Running
--------------- ----------- ----------- --
Dairy Products  4374        2007        1
Confections     4137        2007        2
Beverages       3996        2007        3
Beverages       3694        2008        4
Seafood         3679        2007        5
Condiments      2895        2007        6
Seafood         2716        2008        7
Dairy Products  2689        2008        8
Grains/Cereals  2636        2007        9
```

The next topics will build on this basic use of OVER to define a window of rows.

For further reading on the OVER clause, go to Books Online at:

OVER Clause (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402778

Partitioning Windows

Partitioning a window limits a set to rows with the same value in the partitioning column.

For example, the following code snippet shows the use of PARTITION BY to create a window partition by category. In this example, a partition contains only rows with a category of beverages, or a category of confections:

- Partitioning limits a set to rows with the same value in the partitioning column
- Use PARTITION BY in the OVER() clause
- Without a PARTITION BY clause defined, OVER() creates a single partition of all rows

```
SELECT custid, ordermonth, qty,
       SUM(qty) OVER(PARTITION BY custid)
       AS totalbycust
FROM Sales.CustOrders;
```

```
custid ordermonth               qty totalbycust
------ ----------------------- --- -----------
1      2007-08-01 00:00:00.000 38  174
1      2007-10-01 00:00:00.000 41  174
2      2006-09-01 00:00:00.000 6   63
2      2007-08-01 00:00:00.000 18  63
3      2006-11-01 00:00:00.000 24  359
3      2007-04-01 00:00:00.000 30  359
```

PARTITION BY Code Snippet

```
<function_name>() OVER(PARTITION BY
Category)
```

As you have learned, if no partition is defined, then the OVER() clause returns all rows from the underlying query's result set to the window function.

The following example builds on the one you saw in the previous topic. It adds a PARTITION BY subclause to the OVER clause, creating a window partition for rows with matching Category values. This allows the ROW_NUMBER function to number each set of years per category separately. Note that an ORDER BY subclause has been added to the OVER clause to provide meaning to ROW_NUMBER:

PARTITION BY Example

```
SELECT Category, Qty, Orderyear,
      ROW_NUMBER() OVER (PARTITION BY Category ORDER BY Qty DESC) AS Running
FROM Sales.CategoryQtyYear
ORDER BY Category;
```

The partial result:

```
Category        Qty         Orderyear   Running
--------------- ----------- ----------- ---
Beverages       3996        2007        1
Beverages       3694        2008        2
Beverages       1842        2006        3
Condiments      2895        2007        1
Condiments      1441        2008        2
Condiments      962         2006        3
Confections     4137        2007        1
Confections     2412        2008        2
Confections     1357        2006        3
```

Note: If you intend to add framing to the window partition, an ORDER BY subclause will also be needed in the OVER clause, as discussed in the next topic.

MCT USE ONLY STUDENT USE PROHIBITED

Ordering and Framing

As you have learned, window partitions allow you to define a subset of rows within the outer window defined by OVER. In a similar approach, window framing allows you to further restrict the rows available to the window function. You can think of a frame as a moving window over the data, starting and ending at positions you define.

To define window frames, use the ROW or RANGE subclauses to provide a starting and an ending boundary. For example, to set a frame that extends from the first row in the partition to the current row (such as to create a moving window for a running total), follow these steps:

1. Define an OVER clause with a PARTITION BY element.
2. Define an ORDER BY subclause to the OVER clause. This will cause the concept of "first row" to be meaningful.
3. Add the ROWS BETWEEN subclause, setting the starting boundary using UNBOUNDED PRECEDING. UNBOUNDED means go all the way to the boundary in the direction specified as PRECEDING (before). Add the CURRENT ROW element to indicate the ending boundary is the row being calculated.

Note: Since OVER returns a set, and sets have no order, an ORDER BY subclause is required for the framing operation to be useful. This can be (and typically is) different from ORDER BY, which determines the display order for the final result set.

The following example uses framing to create a moving window, where each row is the end of a frame, starting with the first row in the window partitioned by category and ordered by year. The SUM function calculates an aggregate in each window partition's frame:

Framing Example

```
SELECT Category, Qty, Orderyear,
      SUM(Qty) OVER (PARTITION BY Category ORDER BY Orderyear
             ROWS BETWEEN UNBOUNDED PRECEDING
             AND CURRENT ROW) AS RunningQty
FROM Sales.CategoryQtyYear;
```

The partial results:

```
Category        Qty         Orderyear   RunningQty
--------------- ----------- ----------- -----------
Beverages       1842        2006        1842
Beverages       3996        2007        5838
Beverages       3694        2008        9532
Condiments      962         2006        962
Condiments      2895        2007        3857
Condiments      1441        2008        5298
Confections     1357        2006        1357
Confections     4137        2007        5494
Confections     2412        2008        7906
Dairy Products  2086        2006        2086
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
Dairy Products  4374        2007        6460
Dairy Products  2689        2008        9149
```

Demonstration: Using OVER and Partitioning

In this demonstration, you will see how to:

- Use OVER, PARTITION BY, and ORDER BY clauses.

Demonstration Steps

Use OVER, PARTITION BY, and ORDER BY Clauses

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod13\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod13\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

Sequencing Activity

Put the following elements into the logical order in which they a processed in a windowed query by numbering each to indicate the correct order.

	Steps
	The OVER clause
	The PARTITION BY clause
	The ROW or RANGE clause

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Exploring Window Functions

SQL Server 2014 provides window functions to operate on a window of rows. In addition to window aggregate functions, which you will find to be conceptually similar to grouped aggregate functions, you can use window ranking, distribution, and offset functions in your queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that use window aggregate functions.
- Write queries that use window ranking functions.
- Write queries that use window offset functions.

Defining Window Functions

A window function is applied to a window, or set, of rows. Earlier in this course, you learned about group aggregate functions such as SUM, MIN, and MAX, which operated on a set of rows defined by a GROUP BY clause. In window operations, you can use these functions, as well as others, to operate on a set of rows defined in a window by an OVER clause and its elements.

- A window function is a function applied to a window, or set, of rows
- Window functions include aggregate, ranking, distribution, and offset functions
- Window functions depend on set created by OVER()

```
SELECT  productid, productname, unitprice,
        RANK() OVER(ORDER BY unitprice DESC)
                AS pricerank
FROM Production.Products
ORDER BY pricerank;
```

SQL Server window functions can be found in the following categories, which will be discussed in the next topics:

- Aggregate functions, such as SUM, which operate on a window and return a single row.
- Ranking functions, such as RANK, which depend on a sort order and return a value representing the rank of a row, with respect to other rows in the window.
- Distribution functions, such as CUME_DIST, which calculate the distribution of a value in a window of rows.
- Offset functions, such as LEAD, which return values from other rows relative to the position of the current row.

When used in windowing scenarios, these functions depend on the result set returned by the OVER clause and any further restrictions you provide within OVER, such as partitioning and framing.

The following example uses the RANK function to calculate a rank of each row by unitprice, from high to low value. Note that there is no explicit window partition clause defined:

RANK Example

```
SELECT  productid, productname, unitprice,
        RANK() OVER(ORDER BY unitprice DESC) AS pricerank
FROM Production.Products
ORDER BY pricerank;
```

MCT USE ONLY. STUDENT USE PROHIBITED

The partial result:

```
productid   productname   unitprice             pricerank
----------- ------------- --------------------- ---------
38          Product QDOMO 263.50                1
29          Product VJXYN 123.79                2
9           Product AOZBW 97.00                 3
20          Product QHFFP 81.00                 4
18          Product CKEDC 62.50                 5
59          Product UKXRI 55.00                 6
51          Product APITJ 53.00                 7
62          Product WUXYK 49.30                 8
43          Product ZZZHR 46.00                 9
28          Product OFBNT 45.60                 10
27          Product SMIOH 43.90                 11
63          Product ICKNK 43.90                 11
8           Product WVJFP 40.00                 13
```

For comparison, the following example adds a partition on categoryid (and adds categoryid to the final ORDER BY clause). Note that the ranking is calculated per partition:

RANK with PARTITION Example

```
SELECT  categoryid, productid, unitprice,
        RANK() OVER(PARTITION BY categoryid ORDER BY unitprice DESC) AS pricerank
FROM Production.Products
ORDER BY categoryid, pricerank;
```

The partial result, edited for space:

```
categoryid  productid   unitprice pricerank
----------- ----------- --------- ---------
1           38          263.50    1
1           43          46.00     2
1           2           19.00     3
2           63          43.90     1
2           8           40.00     2
2           61          28.50     3
2           6           25.00     4
3           20          81.00     1
3           62          49.30     2
3           27          43.90     3
3           26          31.23     4
```

Notice that the addition of partitioning allows the window function to operate at a more granular level than when OVER returns an unrestricted set.

Note: Repeating values and gaps in the pricerank column are expected when using RANK in case of ties. Use DENSE_RANK if gaps are not desired. See the next topics for more information.

MCT USE ONLY. STUDENT USE PROHIBITED

Window Aggregate Functions

Window aggregate functions are similar to the aggregate functions you have already used in this course. They aggregate a set of rows and return a single value. However, when used in the context of windows, they operate on the set returned by the OVER clause, not on a set defined by a grouped query using GROUP BY.

- Similar to grouped aggregate functions
  - SUM, MIN, MAX, and so on
- Applied to windows defined by OVER clause
- Window aggregate functions support partitioning, ordering, and framing

```
SELECT  custid, ordermonth, qty,
        SUM(qty) OVER(PARTITION BY custid)
        AS totalpercust
FROM Sales.CustOrders;
```

Window aggregate functions provide support for windowing elements you have learned about in this module, such as partitioning, ordering, and framing. Unlike other window functions, ordering is not required with aggregate functions, unless you are also specifying a frame.

The following example uses a SUM function to return the total sales per customer, displayed as a new column:

Window Aggregate Example

```
SELECT  custid,
        ordermonth,
        qty,
        SUM(qty) OVER ( PARTITION BY custid ) AS totalpercust
FROM    Sales.CustOrders;
```

The partial result, edited for space:

```
custid      ordermonth              qty         totalpercust
----------- ----------------------- ----------- ------------
1           2007-08-01 00:00:00.000 38          174
1           2007-10-01 00:00:00.000 41          174
1           2008-01-01 00:00:00.000 17          174
2           2006-09-01 00:00:00.000 6           63
2           2007-08-01 00:00:00.000 18          63
3           2006-11-01 00:00:00.000 24          359
3           2007-04-01 00:00:00.000 30          359
3           2007-05-01 00:00:00.000 80          359
4           2007-02-01 00:00:00.000 40          650
4           2007-06-01 00:00:00.000 96          650
```

While the repeating of the sum may not immediately seem useful, you can use any manipulation with the result of the window aggregate, such as determining ratios of each sale to the total per customer:

Further Window Aggregate Example

```
SELECT  custid, ordermonth, qty,
        SUM(qty) OVER ( PARTITION BY custid ) AS custtotal,
        CAST(100. * qty/SUM(qty) OVER ( PARTITION BY custid )AS NUMERIC(8,2)) AS
             OfTotal
FROM    Sales.CustOrders;
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The result:

```
custid ordermonth              qty custtotal  OfTotal
------ ----------------------- --- ---------- -------
1      2007-08-01 00:00:00.000 38  174         21.84
1      2007-10-01 00:00:00.000 41  174         23.56
1      2008-01-01 00:00:00.000 17  174         9.77
1      2008-03-01 00:00:00.000 18  174         10.34
1      2008-04-01 00:00:00.000 60  174         34.48
2      2006-09-01 00:00:00.000 6   63          9.52
2      2007-08-01 00:00:00.000 18  63          28.57
2      2007-11-01 00:00:00.000 10  63          15.87
2      2008-03-01 00:00:00.000 29  63          46.03
3      2006-11-01 00:00:00.000 24  359         6.69
3      2007-04-01 00:00:00.000 30  359         8.36
3      2007-05-01 00:00:00.000 80  359         22.28
3      2007-06-01 00:00:00.000 83  359         23.12
3      2007-09-01 00:00:00.000 102 359         28.41
3      2008-01-01 00:00:00.000 40  359         11.14
```

Window Ranking Functions

Window ranking functions return a value representing the rank of a row with respect to other rows in the window. To accomplish this, ranking functions require an ORDER BY element within the OVER clause, to establish the position of each row within the window.

- Ranking functions require a window order clause
  - Partitioning is optional
  - To display results in sorted order still requires ORDER BY!

Function	Description
RANK	Returns the rank of each row within the partition of a result set. May include ties and gaps.
DENSE_RANK	Returns the rank of each row within the partition of a result set. May include ties. Will not include gaps.
ROW_NUMBER	Returns a unique sequential row number within partition based on current order.
NTILE	Distributes the rows in an ordered partition into a specified number of groups. Returns the number of the group to which the current row belongs.

Note: Remember that the ORDER BY element within the OVER clause affects only the processing of rows by the window function. To control the display order of the results, add an ORDER BY clause to the end of the SELECT statement, as with other queries.

The primary difference between RANK and DENSE_RANK is the handling of rows when there are tie values.

For example, the following query uses RANK and DENSE_RANK side-by-side to illustrate how RANK inserts a gap in the numbering after a set of tied row values, whereas DENSE_RANK does not:

RANK and DENSE_RANK Example

```
SELECT CatID, CatName, ProdName, UnitPrice,
      RANK() OVER(PARTITION BY CatID ORDER BY UnitPrice DESC) AS PriceRank,
      DENSE_RANK() OVER(PARTITION BY CatID ORDER BY UnitPrice DESC) AS DensePriceRank
FROM Production.CategorizedProducts
ORDER BY CatID;
```

The partial results follow. Note the rank numbering of the rows following the products with a unitprice of 18.00:

```
CatID CatName   ProdName      UnitPrice PriceRank DensePriceRank
----- --------- ------------- --------- --------- --------------
1     Beverages Product QDOMO 263.50    1         1
1     Beverages Product ZZZHR 46.00     2         2
1     Beverages Product RECZE 19.00     3         3
1     Beverages Product HHYDP 18.00     4         4
1     Beverages Product LSOFL 18.00     4         4
1     Beverages Product NEVTJ 18.00     4         4
1     Beverages Product JYGFE 18.00     4         4
1     Beverages Product TOONT 15.00     8         5
1     Beverages Product XLXQF 14.00     9         6
1     Beverages Product SWNJY 14.00     9         6
1     Beverages Product BWRLG 7.75      11        7
1     Beverages Product QOGNU 4.50      12        8
```

Go to *Ranking Functions (Transact-SQL)* in Books Online at:

Ranking Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402779

Window Distribution Functions

Window distribution functions perform statistical analysis on the rows within the window or window partition. Partitioning a window is optional for distribution functions, but ordering is required.

- Window distribution functions perform statistical analysis on data, and require a window order clause
- Rank distribution performed with PERCENT_RANK and CUME_DIST
- Inverse distribution performed with PERCENTILE_CONT and PERCENTILE_DISC

Distribution functions return a rank of a row, but instead of being expressed as an ordinal number, as with RANK, DENSE_RANK, or ROW_NUMBER, it is expressed as a ratio between 0 and 1. SQL Server 2014 provides rank distribution with the PERCENT_RANK and CUME_DIST functions. It provides inverse distribution with the PERCENTILE_CONT and PERCENTILE_DISC functions.

These functions are listed here for completeness only and are beyond the scope of this course. For more information, go to Books Online at:

Analytic Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402780

MCT USE ONLY. STUDENT USE PROHIBITED

Window Offset Functions

Windows offset functions enable access to values located in rows other than the current row. This can enable queries that perform comparisons between rows, without the need to join the table to itself.

Offset functions operate on a position that is either relative to the current row, or relative to the starting or ending boundary of the window frame. LAG and LEAD operate on an offset to the current row. FIRST_VALUE and LAST_VALUE operate on an offset from the window frame.

- Window offset functions allow comparisons between rows in a set without the need for a self-join
- Offset functions operate on a position relative to the current row, or to the start or end of the window frame

Function	Description
LAG	Returns an expression from a previous row that is a defined offset from the current row. Returns NULL if no row at specified position.
LEAD	Returns an expression from a later row that is a defined offset from the current row. Returns NULL if no row at specified position.
FIRST_VALUE	Returns the first value in the current window frame. Requires window ordering to be meaningful.
LAST_VALUE	Returns the last value in the current window frame. Requires window ordering to be meaningful.

Note: Since FIRST_VALUE and LAST_VALUE operate on offsets from the window frame, it is important to remember to specify framing options other than the default of RANGE BETWEEN UNBOUND PRECEDING AND CURRENT ROW.

The following example uses the LEAD function to compare year-over-year sales. The offset is 1, returning the next row's value. LEAD returns a 0 if a NULL is found in the next row's value, such as when there are no sales past the latest year:

Window Offset Function Example

```
SELECT employee, orderyear ,totalsales AS currsales,
   LEAD(totalsales, 1,0) OVER (PARTITION BY employee ORDER BY orderyear) AS nextsales
FROM Sales.OrdersByEmployeeYear
ORDER BY employee, orderyear;
```

The partial results:

```
employee orderyear currsales nextsales
-------- --------- --------- ---------
1           2006   38789.00  97533.58
1           2007   97533.58  65821.13
1           2008   65821.13  0.00
2           2006   22834.70  74958.60
2           2007   74958.60  79955.96
2           2008   79955.96  0.00
3           2006   19231.80 111788.61
3           2007  111788.61  82030.89
3           2008   82030.89  0.00
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Exploring Windows Functions

In this demonstration, you will see how to:

- Use window aggregate, ranking, and offset functions.

Demonstration Steps

Use Window Aggregate, Ranking, and Offset Functions

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod13\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod13\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

Categorize Activity

Categorize each windowing function into the appropriate category. Indicate your answer by writing the category number to the right of each item.

Items	
1	MIN()
2	DENSERANK()
3	PERCENTILE_DISC()
4	PERCENTILE_CONT()
5	NTITLE()
6	RANK()
7	SUM()
8	MAX()
9	ROW_NUMBER()
10	CUME_DIST()
11	PERCENT_RANK()

Category 1		Category 2		Category 3
Window Aggregate Functions		Window Ranking Functions		Window Distribution Functions

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Using Window Ranking, Offset, and Aggregate Functions

Scenario

You are a business analyst for Adventure Works, who will be writing reports using corporate databases stored in SQL Server 2014. You have been provided with a set of business requirements for data and you will write T-SQL queries to retrieve the specified data from the databases. To fill these requests, you will need to calculate ranking values, as well as the difference between two consecutive rows, and running totals. You will use window functions to achieve these calculations.

Objectives

After completing this lab, you will be able to:

- Write queries that use ranking functions.
- Write queries that use offset functions.
- Write queries that use window aggregation functions.

Estimated Time: 60 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Writing Queries That Use Ranking Functions

Scenario

The sales department would like to rank orders by their values for each customer. You will provide the report by using the RANK function. You will also practice how to add a calculated column to display the row number in the SELECT clause.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment

2. Write a SELECT Statement That Uses the ROW_NUMBER Function to Create a Calculated Column

3. Add an Additional Column Using the RANK Function

4. Write A SELECT Statement to Calculate a Rank, Partitioned by Customer and Ordered by the Order Value

5. Write a SELECT Statement to Rank Orders, Partitioned by Customer and Order Year, and Ordered by the Order Value

6. Filter Only Orders with the Top Two Ranks

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.

2. Run Setup.cmd in the D:\Labfiles\Lab13\Starter folder as Administrator.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Write a SELECT Statement That Uses the ROW_NUMBER Function to Create a Calculated Column

1. In SQL Server Management Studio, open the project file D:\Labfiles\Lab13\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the orderid, orderdate, and val columns as well as a calculated column named rowno from the view Sales.OrderValues. Use the ROW_NUMBER function to return rowno. Order the row numbers by the orderdate column.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.

▶ Task 3: Add an Additional Column Using the RANK Function

1. Copy the previous T-SQL statement and modify it by including an additional column named rankno. To create rankno, use the RANK function, with the rank order based on the orderdate column.
2. Execute the modified statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\ Solution\53 - Lab Exercise 1 - Task 2 Result.txt. Notice the different values in the rowno and rankno columns for some of the rows.
3. What is the difference between the RANK and ROW_NUMBER functions?

▶ Task 4: Write A SELECT Statement to Calculate a Rank, Partitioned by Customer and Ordered by the Order Value

1. Write a SELECT statement to retrieve the orderid, orderdate, custid, and val columns, as well as a calculated column named orderrankno from the Sales.OrderValues view. The orderrankno column should display the rank per each customer independently, based on val ordering in descending order.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\ Solution\54 - Lab Exercise 1 - Task 3 Result.txt.

▶ Task 5: Write a SELECT Statement to Rank Orders, Partitioned by Customer and Order Year, and Ordered by the Order Value

1. Write a SELECT statement to retrieve the custid and val columns from the Sales.OrderValues view. Add two calculated columns:
   - orderyear as a year of the orderdate column.
   - orderrankno as a rank number, partitioned by the customer and order year, and ordered by the order value in descending order.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\ Solution\55 - Lab Exercise 1 - Task 4 Result.txt.

▶ Task 6: Filter Only Orders with the Top Two Ranks

1. Copy the previous query and modify it to filter only orders with the first two ranks based on the orderrankno column.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\ Solution\56 - Lab Exercise 1 - Task 5 Result.txt.

Results: After this exercise, you should know how to use ranking functions in T-SQL statements.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Writing Queries That Use Offset Functions

Scenario

You need to provide separate reports to analyze the difference between two consecutive rows. This will enable business users to analyze growth and trends.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Next Row Using a Common Table Expression (CTE)
2. Add a Column to Display the Running Sales Total
3. Analyze the Sales Information for the Year 2007

▶ Task 1: Write a SELECT Statement to Retrieve the Next Row Using a Common Table Expression (CTE)

1. Open the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. Define a CTE named OrderRows based on a query that retrieves the orderid, orderdate, and val columns from the Sales.OrderValues view. Add a calculated column named rowno using the ROW_NUMBER function, ordering by the orderdate and orderid columns.
3. Write a SELECT statement against the CTE and use the LEFT JOIN with the same CTE to retrieve the current row and the previous row based on the rowno column. Return the orderid, orderdate, and val columns for the current row and the val column from the previous row as prevval. Add a calculated column named diffprev to show the difference between the current val and previous val.
4. Execute the T-SQL code and compare the results that you got with the desired results shown in the file D:\Labfiles\Lab13\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Add a Column to Display the Running Sales Total

1. Write a SELECT statement that uses the LAG function to achieve the same results as the query in the previous task. The query should not define a CTE.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Analyze the Sales Information for the Year 2007

1. Define a CTE named SalesMonth2007 that creates two columns: monthno (the month number of the orderdate column) and val (aggregated val column). Filter the results to include only the order year 2007 and group by monthno.
2. Write a SELECT statement to retrieve the monthno and val columns. Add two calculated columns:
   - avglast3months. This column should contain the average sales amount for the last three months before the current month, using a window aggregate function. You can assume that there are no missing months.
   - ytdval. This column should contain the cumulative sales value up to the current month.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\63 - Lab Exercise 2 - Task 3 Result.txt.

Results: After this exercise, you should be able to use the offset functions in your T-SQL statements.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Writing Queries That Use Window Aggregate Functions

Scenario

To better understand the cumulative sales value of a customer through time and to provide the sales analyst with a year-to-date analysis, you will have to write different SELECT statements that use the window aggregate functions.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Display the Contribution of Each Customer's Order Compared to That Customer's Total Purchase

2. Add a Column to Display the Running Sales Total

3. Analyze the Year-to-Date Sales Amount and Average Sales Amount for the Last Three Months

▶ Task 1: Write a SELECT Statement to Display the Contribution of Each Customer's Order Compared to That Customer's Total Purchase

1. Open the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the custid, orderid, orderdate, and val columns from the Sales.OrderValues view. Add a calculated column named percoftotalcust containing a percentage value of each order sales amount compared to the total sales amount for that customer.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Add a Column to Display the Running Sales Total

1. Copy the previous SELECT statement and modify it by adding a new calculated column named runval. This column should contain a running sales total for each customer based on order date, using orderid as the tiebreaker.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

▶ Task 3: Analyze the Year-to-Date Sales Amount and Average Sales Amount for the Last Three Months

1. Copy the SalesMonth2007 CTE in the last task in exercise 2. Write a SELECT statement to retrieve the monthno and val columns. Add two calculated columns:
   - avglast3months. This column should contain the average sales amount for the last three months before the current month using a window aggregate function. You can assume that there are no missing months.
   - ytdval. This column should contain the cumulative sales value up to the current month.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\74 - Lab Exercise 3 - Task 3 Result.txt.

Results: After this exercise, you should have a basic understanding of how to use window aggregate functions in T-SQL statements.

Module Review and Takeaways

Review Question(s)

Question: What results will be returned by a ROW_NUMBER function if there is no ORDER BY clause in the query?

Question: Which ranking function would you use to return the values 1,1,3? Which would return 1,1,2?

Question: Can a window frame extend beyond the boundaries of the window partition defined in the same OVER() clause?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 14

Pivoting and Grouping Sets

Contents:

Module Overview	**14-1**
Lesson 1: Writing Queries with PIVOT and UNPIVOT	**14-2**
Lesson 2: Working with Grouping Sets	**14-7**
Lab: Pivoting and Grouping Sets	**14-13**
Module Review and Takeaways	**14-19**

Module Overview

This module discusses more advanced manipulations of data, building on the basics you have learned so far in the course. First, you will learn how to use the PIVOT and UNPIVOT operators to change the orientation of data from column-oriented to row-oriented and back. Next, you will learn how to use the GROUPING SET subclause of the GROUP BY clause to specify multiple groupings in a single query. This will include the use of the CUBE and ROLLUP subclauses of GROUP BY to automate the setup of grouping sets.

Objectives

After completing this module, you will be able to:

- Write queries that pivot and unpivot result sets.
- Write queries that specify multiple groupings with grouping sets.

Lesson 1
Writing Queries with PIVOT and UNPIVOT

Sometimes you may need to present data in a different orientation to how it is stored, with respect to row and column layout. For example, some data may be easier to compare if you can arrange values across columns of the same row. In this lesson, you will learn how to use the T-SQL PIVOT operator to accomplish this. You will also learn how to use the UNPIVOT operator to return the data to a rows-based orientation.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how pivoting data can be used in T-SQL queries.
- Write queries that pivot data from rows to columns using the PIVOT operator.
- Write queries that unpivot data from columns back to rows using the UNPIVOT operator.

What Is Pivoting?

Pivoting data in SQL Server rotates its display from a rows-based orientation to a columns-based orientation. It does this by consolidating values in a column to a list of distinct values, and then projects that list across as column headings. Typically this includes aggregation to column values in the new columns.

For example, the partial source data below lists repeating values for Category and Orderyear, along with values for Qty, for each instance of a Category/Orderyear pair:

```
Category        Qty    Orderyear
--------------- ------ -----------
Dairy Products  12     2006
Grains/Cereals  10     2006
Dairy Products  5      2006
Seafood         2      2007
Confections     36     2007
Condiments      35     2007
Beverages       60     2007
Confections     55     2007
Condiments      16     2007
Produce         15     2007
Dairy Products  60     2007
Dairy Products  20     2007
Confections     24     2007
...
Condiments      2      2008
 (2155 row(s) affected)
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

To analyze this by category and year, you may want to arrange the values to be displayed as follows, summing the Qty column along the way:

```
Category       2006 2007 2008
-------------- ---- ---- ----
Beverages      1842 3996 3694
Condiments     962  2895 1441
Confections    1357 4137 2412
Dairy Products 2086 4374 2689
Grains/Cereals 549  2636 1377
Meat/Poultry   950  2189 1060
Produce        549  1583 858
Seafood        1286 3679 2716
 (8 row(s) affected)
```

In the pivoting process, each distinct year was created as a column header, and values in the Qty column were grouped by Category and aggregated. This is a very useful technique in many scenarios.

Additional reading can be found in Books Online at:

Using PIVOT and UNPIVOT

http://go.microsoft.com/fwlink/?LinkID=402781

Elements of PIVOT

The T-SQL PIVOT table operator, introduced in Microsoft® SQL Server® 2005, operates on the output of the FROM clause in a SELECT statement. To use PIVOT, you need to supply three elements to the operator:

- Pivoting includes three phases:
  1. Grouping determines which element gets a row in the result set
  2. Spreading provides the distinct values to be pivoted across
  3. Aggregation performs an aggregation function (such as SUM)

- **Grouping**: In the FROM clause, you need to provide the input columns. From those columns, PIVOT will determine which column(s) will be used to group the data for aggregation. This is based on looking at which columns are not being used as other elements in the PIVOT operator.
- **Spreading**: You need to provide a comma-delimited list of values to be used as the column headings for the pivoted data. The values need to occur in the source data.
- **Aggregation**: You need to provide an aggregation function (SUM, and so on) to be performed on the grouped rows.

Additionally, you need to assign a table alias to the result table of the PIVOT operator. The following example shows the elements in place:

In this example, Orderyear is the column providing the spreading values, Qty is used for aggregation, and Category for grouping. Orderyear values are enclosed in delimiters to indicate that they are identifiers of columns in the result:

PIVOT Example

```
SELECT  Category, [2006],[2007],[2008]
FROM  ( SELECT  Category, Qty, Orderyear FROM Sales.CategoryQtyYear) AS D
        PIVOT(SUM(qty) FOR orderyear IN ([2006],[2007],[2008])) AS pvt;
```

Note: Any attributes in the source subquery, not used for aggregation or spreading, will be used as grouping elements, so be sure that no unnecessary attributes are included in the subquery.

One of the challenges in writing queries using PIVOT is the need to supply a fixed list of spreading elements to the PIVOT operator, such as the specific order year values above. Later in this course, you will learn how to write dynamically-generated queries, which may help you write PIVOT queries with more flexibility.

Writing Queries with UNPIVOT

Unpivoting data is the logical reverse of pivoting data. Instead of turning rows into columns, unpivot turns columns into rows. This is a technique useful in taking data that has already been pivoted (with or without using a T-SQL PIVOT operator) and returning it to a row-oriented tabular display. SQL Server provides the UNPIVOT table operator to accomplish this.

- Unpivoting data is rotating data from a columns-based orientation to a rows-based orientation
- Spreads or splits values from one source row into one or more target rows
- Each source row becomes one or more rows in result set based on number of columns being pivoted
- Unpivoting includes three elements:
  - Source columns to be unpivoted
  - Name to be assigned to new values column
  - Name to be assigned to names columns

When unpivoting data, one or more columns is defined as the source to be converted into rows. The data in those columns is spread, or split, into one or more new rows, depending on how many columns are being unpivoted.

In the following source data, three columns will be unpivoted. Each Orderyear value will be copied into a new row and associated with its Category value. Any NULLs will be removed in the process and no row is created:

```
Category        2006 2007 2008
--------------- ---- ---- ----
Beverages       1842 3996 3694
Condiments      962  2895 1441
Confections     1357 4137 2412
Dairy Products  2086  374 2689
Grains/Cereals  549  2636 1377
Meat/Poultry    950  2189 1060
Produce         549  1583 858
Seafood         1286 3679 2716
```

For each intersection of Category and Orderyear, a new row will be created, as in these partial results:

```
category        qty  orderyear
--------------- ---- ---------
Beverages       1842 2006
Beverages       3996 2007
Beverages       3694 2008
Condiments      962  2006
Condiments      2895 2007
Condiments      1441 2008
Confections     1357 2006
Confections     4137 2007
Confections     2412 2008
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Note: Unpivoting does not restore the original data. Detail-level data was lost during the aggregation process in the original pivot. UNPIVOT has no ability to allocate values to return to original detail values.

To use the UNPIVOT operator, you need to provide three elements:

- Source columns to be unpivoted.
- A name for the new column that will display the unpivoted values.
- A name for the column that will display the names of the unpivoted values.

Note: As with PIVOT, you will define the output of the UNPIVOT table operator as a derived table and provide its name.

The following example specifies 2006, 2007, and 2008 as the columns to be unpivoted, using the new column name orderyear and the qty values to be displayed in a new qty column. (This technique was used to generate the sample data in the previous example):

UNPIVOT Example

```
SELECT category, qty, orderyear
FROM Sales.PivotedCategorySales
UNPIVOT(qty FOR orderyear IN([2006],[2007],[2008])) AS unpvt;
```

The partial results:

```
category        qty         orderyear
--------------- ----------- ---------
Beverages       1842        2006
Beverages       3996        2007
Beverages       3694        2008
Condiments      962         2006
Condiments      2895        2007
Condiments      1441        2008
Confections     1357        2006
Confections     4137        2007
Confections     2412        2008
Dairy Products  2086        2006
Dairy Products  4374        2007
Dairy Products  2689        2008
```

Demonstration: Writing Queries with PIVOT and UNPIVOT

In this demonstration, you will see how to:

- Use PIVOT and UNPIVOT.

Demonstration Steps

Use PIVOT and UNPIVOT

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod14\Setup.cmd as an administrator.

3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod14\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

Question: You have the following query:

```
SELECT category, qty, orderyear
FROM Sales.PivotedCategorySales
UNPIVOT(qty FOR orderyear) AS unpvt;
```

In this query, you have provided a name for the new column that will display the unpivoted values ("qty"). You have also provided a name for the column that will display the names of the unpivoted values (orderyear). What else must you provide for the UNPIVOT query to execute?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Working with Grouping Sets

As you learned earlier in this course, you can use the GROUP BY clause in a SELECT statement to arrange rows in groups, typically to support aggregations. However, if you need to group by different attributes at the same time, for example to report at different levels, you will need multiple queries combined with UNION ALL. SQL Server 2008 and later provides the GROUPING SETS subclause to GROUP BY, which enables multiple sets to be returned in the same query.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries using the GROUPING SETS subclause.
- Write queries that use ROLLUP AND CUBE.
- Write queries that use the GROUPING_ID function.

Writing Queries with Grouping Sets

If you need to produce aggregates of multiple groupings in the same query, you can use the GROUPING SETS subclause of the GROUP BY clause.

GROUPING SETS provide an alternative to using UNION ALL to combine results from multiple individual queries, each with its own GROUP BY clause.

With GROUPING SETS, you can specify multiple combinations of attributes on which to group, as in the following syntax example:

- GROUPING SETS subclause builds on T-SQL GROUP BY clause
- Allows multiple groupings to be defined in same query
- Alternative to use of UNION ALL to combine multiple outputs (each with different GROUP BY) into one result set

```
SELECT <column list with aggregate(s)>
FROM <source>
GROUP BY
GROUPING SETS(
        (<column_name>),--one or more columns
        (<column_name>),--one or more columns
        () -- empty parentheses if aggregating all rows
              );
```

GROUPING SETS Syntax

```
SELECT <column list with aggregate(s)>
FROM <source>
GROUP BY
GROUPING SETS(
      (<column_name>),--one or more columns
      (<column_name>),--one or more columns
      () -- empty parentheses if aggregating all rows
             );
```

With GROUPING SETS, you can specify which attributes to group on and their order. If you want to group on any possible combination of attributes instead, see the topic on CUBE and ROLLUP later in this lesson.

The following example uses GROUPING SETS to aggregate on the Category and Cust columns, as well as the empty parentheses notation to aggregate all rows:

GROUPING SETS Example

```
SELECT Category, Cust, SUM(Qty) AS TotalQty
FROM Sales.CategorySales
GROUP BY
      GROUPING SETS((Category),(Cust),())
ORDER BY Category, Cust;
```

The results:

```
Category    Cust TotalQty
----------- ---- --------
NULL        NULL 999
NULL        1    80
NULL        2    12
NULL        3    154
NULL        4    241
NULL        5    512
Beverages   NULL 513
Condiments  NULL 114
Confections NULL 372
```

Note the presence of NULLs in the results. NULLs may be returned because a NULL was stored in the underlying source, or because it is a placeholder in a row generated as an aggregate result. For example, in the previous results, the first row displays NULL, NULL, 999. This represents a grand total row. The NULL in the Category and Cust columns are placeholders since neither Category nor Cust take part in the aggregation.

Go to Using GROUP BY with ROLLUP, CUBE, and GROUPING SETS in Books Online at:

Using GROUP BY with ROLLUP, CUBE, and GROUPING SETS

http://go.microsoft.com/fwlink/?LinkID=402782

CUBE and ROLLUP

Like GROUPING SETS, the CUBE and ROLLUP subclauses also enable multiple groupings for aggregating data. However, CUBE and ROLLUP do not need you to specify each set of attributes to group. Instead, given a set of columns, CUBE will determine all possible combinations and output groupings. ROLLUP creates combinations, assuming the input columns represent a hierarchy. Therefore, CUBE and ROLLUP can be thought of as shortcuts to GROUPING SETS.

To use CUBE, append the keyword CUBE to the GROUP BY clause and provide a list of columns to group.

- CUBE provides shortcut for defining grouping sets given a list of columns
- All possible combinations of grouping sets created

```
SELECT Category, Cust, SUM(Qty) AS TotalQty
FROM Sales.CategorySales
GROUP BY CUBE(Category,Cust)
ORDER BY Category, Cust;
```

- ROLLUP provides shortcut for defining grouping sets, creates combinations assuming input columns form a hierarchy

```
SELECT Category, Cust, SUM(Qty) AS TotalQty
FROM Sales.CategorySales
GROUP BY ROLLUP(Category,Cust)
ORDER BY Category, Cust;
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

For example, to group on all combinations of columns Category and Cust, use the following syntax in your query:

CUBE Example

```
SELECT Category, Cust, SUM(Qty) AS TotalQty
FROM Sales.CategorySales
GROUP BY CUBE(Category,Cust);
```

This will output groupings for the following combinations: (Category, Cust), (Cust, Category), (Cust), (Category) and the aggregate on all empty ().

To use ROLLUP, append the keyword ROLLUP to the GROUP BY clause and provide a list of columns to group. For example, to group on combinations of the Category, Subcategory, and Product columns, use the following syntax in your query:

ROLLUP Example

```
SELECT Category, Subcategory, Product, SUM(Qty) AS TotalQty
FROM Sales.ProductSales
GROUP BY ROLLUP(Category,Subcategory, Product);
```

This will output groupings for the following combinations: (Category, Subcategory, Product), (Category, Subcategory), (Category), and the aggregate on all empty (). Note that the order in which columns are supplied is significant: ROLLUP assumes that the columns are listed in an order that expresses a hierarchy.

Note: The example just given is for illustration only. Object names do not correspond to the sample database supplied with the course.

GROUPING_ID

As you have seen, multiple grouping sets allow you to combine different levels of aggregation in the same query. You have also learned that SQL Server will mark placeholder values with NULL if a row does not take part in a grouping set. In a query with multiple sets, however, how do you know whether a NULL marks a placeholder or comes from the underlying data? If it marks a placeholder for a grouping set, which set? The GROUPING_ID function can help you provide additional information to answer these questions.

- Multiple grouping sets present a problem in identifying the source of each row in the result set
- NULLs could come from the source data or could be a placeholder in the grouping set
- The GROUPING_ID function provides a method to mark a row with a 1 or 0 to identify which grouping set the row is a member of

```
SELECT GROUPING_ID(Category) AS grpCat,
       GROUPING_ID(Cust) AS grpCust,
       Category, Cust, SUM(Qty) AS TotalQty
FROM Sales.CategorySales
GROUP BY CUBE(Category,Cust)
ORDER BY Category, Cust;
```

For example, consider the following query and results, which contain numerous NULLs:

Grouping Sets with NULLs Example

```
SELECT Category, Cust, SUM(Qty) AS TotalQty
FROM Sales.CategorySales
GROUP BY
GROUPING SETS((Category),(Cust),())
ORDER BY Category, Cust;
```

The partial results:

```
Category        Cust        TotalQty
--------------- ----------- --------
NULL            NULL        999
NULL            1           80
NULL            2           12
NULL            3           154
NULL            4           241
NULL            5           512
Beverages       NULL        513
Condiments      NULL        114
Confections     NULL        372
```

At a glance, it may be difficult to determine why a NULL appears in a column.

The GROUPING_ID function can be used to associate result rows with their grouping sets, as follows:

GROUPING_ID Example

```
SELECT
      GROUPING_ID(Category)AS grpCat,
      GROUPING_ID(Cust) AS grpCust,
      Category, Cust, SUM(Qty) AS TotalQty
FROM Sales.CategorySales
GROUP BY CUBE(Category,Cust);
```

The partial results:

```
grpCat      grpCust     Category        Cust        TotalQty
----------- ----------- --------------- ----------- -----------
0           0           Beverages       1           36
0           0           Condiments      1           44
1           0           NULL            1           80
0           0           Beverages       2           5
0           0           Confections     2           7
1           0           NULL            2           12
0           0           Beverages       3           105
0           0           Condiments      3           4
0           0           Confections     3           45
1           0           NULL            3           154
...
1           1           NULL            NULL        999
0           1           Beverages       NULL        513
0           1           Condiments      NULL        114
0           1           Confections     NULL        372
```

As you can see, the GROUPING_ID function returns a 1 when a row is aggregated as part of the current grouping set and a 0 when it is not. In the first row, both grpCat and grpCust return 0; therefore, the row is part of the grouping set (Category, Cust).

GROUPING_ID can also take multiple columns as inputs and return a unique integer bitmap, comprised of combined bits, per grouping set. For more information, go to Books Online at:

GROUPING_ID (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402787

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

SQL Server also provides a GROUPING function, which accepts only one input to return a bit. Go to *GROUPING (Transact-SQL)* in Books Online at:

GROUPING (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402788

Demonstration: Using Grouping Sets

In this demonstration, you will see how to:

- Use the CUBE and ROLLUP subclauses

Demonstration Steps

Use the CUBE and ROLLUP Subclauses

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod14\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod14\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

Check Your Knowledge

Question	

You have the following query:
 	
`SELECT e.Department, e.Country, COUNT(EmployeeID) AS Staff`
`FROM HumanResources.Employees AS e`
 	
You want to find out how many staff are in each country and how many staff are in each department. You also want to find out how many staff are in Sales in the US and so on with all departments in all countries where the company operates. Choose the most succinct grouping technique for this query:	

Select the correct answer.	
	GROUPING SETS
	CUBE
	ROLLUP
	You cannot return the required data with GROUPING. Instead, use multiple queries and a UNION element.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Pivoting and Grouping Sets

Scenario

You are a business analyst for Adventure Works, who will be writing reports using corporate databases stored in SQL Server 2014. You have been provided with a set of business requirements for data and you will write T-SQL queries to retrieve the specified data from the databases. The business requests are analytical in nature. To fulfill those requests, you will need to provide crosstab reports and multiple aggregates based on different granularities. Therefore, you will need to use pivoting techniques and grouping sets in your T-SQL code.

Objectives

After completing this lab, you will be able to:

- Write queries that use the PIVOT operator.
- Write queries that use the UNPIVOT operator.
- Write queries that use the GROUPING SETS, CUBE, and ROLLUP subclauses.

Estimated Time: 60 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Writing Queries That Use the PIVOT Operator

Scenario

The sales department would like to have a crosstab report, displaying the number of customers for each customer group and country. They would like to display each customer group as a new column. You will write different SELECT statements using the PIVOT operator to achieve the needed result.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement to Retrieve the Number of Customers for a Specific Customer Group
3. Specify the Grouping Element for the PIVOT Operator
4. Use a Common Table Expression (CTE) to Specify the Grouping Element for the PIVOT Operator
5. Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer and Product Category

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab14\Starter folder as Administrator.

▶ Task 2: Write a SELECT Statement to Retrieve the Number of Customers for a Specific Customer Group

1. In SQL Server Management Studio, open the project file D:\Labfiles\Lab14\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.

2. The IT department has provided you with T-SQL code to generate a view named Sales.CustGroups, which contains three pieces of information about customers—their IDs, the countries in which they are located, and the customer group in which they have been placed. Customers are placed into one of three predefined groups (A, B, or C).
3. Execute the provided T-SQL code:

```
CREATE VIEW Sales.CustGroups AS
SELECT
custid,
CHOOSE(custid % 3 + 1, N'A', N'B', N'C') AS custgroup,
country
FROM Sales.Customers;
```

4. Write a SELECT statement that will return the custid, custgroup, and country columns from the newly created Sales.CustGroups view.
5. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab14\Solution\52 - Lab Exercise 1 - Task 1_1 Result.txt.
6. Modify the SELECT statement. Begin by retrieving the column country then use the PIVOT operator to retrieve three columns based on the possible values of the custgroup column (values A, B, and C), showing the number of customers in each group.
7. Execute the modified statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab14\53 - Lab Exercise 1 - Task 1_2 Result.txt.

▶ Task 3: Specify the Grouping Element for the PIVOT Operator

1. The IT department has provided T-SQL code to add two new columns—city and contactname—to the Sales.CustGroups view. Execute the provided T-SQL code:

```
ALTER VIEW Sales.CustGroups AS
SELECT
custid,
CHOOSE(custid % 3 + 1, N'A', N'B', N'C') AS custgroup,
country,
city,
contactname
FROM Sales.Customers;
```

2. Copy the last SELECT statement in task 1 and execute it.
3. Is this result the same as that from the query in task 1? Is the number of rows retrieved the same?
4. To better understand the reason for the different results, modify the copied SELECT statement to include the new city and contactname columns.
5. Execute the modified statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab14\54 - Lab Exercise 1 - Task 2 Result.txt.
6. Notice that this query returned the same number of rows as the previous SELECT statement. Why did you get the same result with and without specifying the grouping columns for the PIVOT operator?

▶ Task 4: Use a Common Table Expression (CTE) to Specify the Grouping Element for the PIVOT Operator

1. Define a CTE named PivotCustGroups based on a query that retrieves the custid, country, and custgroup columns from the Sales.CustGroups view. Write a SELECT statement against the CTE, using a PIVOT operator to retrieve the same result as in task 1.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

2. Execute the written T-SQL code and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab14\55 - Lab Exercise 1 - Task 3 Result.txt.
3. Is this result the same as the one returned by the last query in task 1? Can you explain why?
4. Why do you think it is beneficial to use the CTE when using the PIVOT operator?

▶ Task 5: Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer and Product Category

1. For each customer, write a SELECT statement to retrieve the total sales amount for all product categories, displaying each as a separate column. Here is how to accomplish this task:
   - Create a CTE named SalesByCategory to retrieve the custid column from the Sales.Orders table as a calculated column, based on the qty and unitprice columns and the categoryname column from the table Production.Categories. Filter the result to include only orders in the year 2008.
   - You will need to JOIN tables Sales.Orders, Sales.OrderDetails, Production.Products, and Production.Categories.
   - Write a SELECT statement against the CTE that returns a row for each customer (custid) and a column for each product category, with the total sales amount for the current customer and product category.
   - Display the following product categories: Beverages, Condiments, Confections, [Dairy Products], [Grains/Cereals], [Meat/Poultry], Produce, and Seafood.
2. Execute the complete T-SQL code (the CTE and the SELECT statement).
3. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab14\56 - Lab Exercise 1 - Task 4 Result.txt.

Results: After this exercise, you should be able to use the PIVOT operator in T-SQL statements.

Exercise 2: Writing Queries That Use the UNPIVOT Operator

Scenario

You will now create multiple rows by turning columns into rows.

The main tasks for this exercise are as follows:

1. Create and Query the Sales.PivotCustGroups View
2. Write a SELECT Statement to Retrieve a Row for Each Country and Customer Group
3. Remove the Created Views

▶ Task 1: Create and Query the Sales.PivotCustGroups View

1. Open the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. Execute the provided T-SQL code to generate the Sales.PivotCustGroups view:

```
CREATE VIEW Sales.PivotCustGroups AS
WITH PivotCustGroups AS
(
SELECT
custid,
country,
custgroup
```

```
FROM Sales.CustGroups
)
SELECT
country,
p.A,
p.B,
p.C
FROM PivotCustGroups
PIVOT (COUNT(custid) FOR custgroup IN (A, B, C)) AS p;
```

3. Write a SELECT statement to retrieve the country, A, B, and C columns from the Sales.PivotCustGroups view.
4. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab14\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Retrieve a Row for Each Country and Customer Group

a. Write a SELECT statement against the Sales.PivotCustGroups view that returns the following:
   - A row for each country and customer group.
   - The column country.
   - Two new columns—custgroup and numberofcustomers. The custgroup column should hold the names of the source columns A, B, and C as character strings, and the numberofcustomers column should hold their values (that is, number of customers).

b. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab14\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Remove the Created Views

1. Remove the created views by executing the provided T-SQL code:

```
DROP VIEW Sales.CustGroups;
DROP VIEW Sales.PivotCustGroups;
```

2. Execute this code exactly as written, inside a query window.

Results: After this exercise, you should know how to use the UNPIVOT operator in your T-SQL statements.

Exercise 3: Writing Queries That Use the GROUPING SETS, CUBE, and ROLLUP Subclauses

Scenario

You have to prepare SELECT statements to retrieve a unified result set with aggregated data for different combinations of columns. First, you have to retrieve the number of customers for all possible combinations of the country and city columns. Instead of using multiple T-SQL statements with a GROUP BY clause and then unifying them with the UNION ALL operator, you will use a more elegant solution using the GROUPING SETS subclause of the GROUP BY clause.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement That Uses the GROUPING SETS Subclause to Return the Number of Customers for Different Grouping Sets

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

2. Write a SELECT Statement That Uses the CUBE Subclause to Retrieve Grouping Sets Based on Yearly, Monthly, and Daily Sales Values

3. Write the Same SELECT Statement Using the ROLLUP Subclause

4. Analyze the Total Sales Value by Year and Month

▶ Task 1: Write a SELECT Statement That Uses the GROUPING SETS Subclause to Return the Number of Customers for Different Grouping Sets

1. Open the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement against the Sales.Customers table and retrieve the country column, the city column, and a calculated column noofcustomers as a count of customers. Retrieve multiple grouping sets based on the country and city columns, the country column, the city column, and a column with an empty grouping set.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab14\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement That Uses the CUBE Subclause to Retrieve Grouping Sets Based on Yearly, Monthly, and Daily Sales Values

1. Write a SELECT statement against the view Sales.OrderValues and retrieve these columns:
   - Year of the orderdate column as orderyear.
   - Month of the orderdate column as ordermonth.
   - Day of the orderdate column as orderday.
   - Total sales value using the val column as salesvalue.
   - Return all possible grouping sets based on the orderyear, ordermonth, and orderday columns.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab14\73 - Lab Exercise 3 - Task 2 Result.txt. Notice the total number of rows in your results.

▶ Task 3: Write the Same SELECT Statement Using the ROLLUP Subclause

1. Copy the previous query and modify it to use the ROLLUP subclause instead of the CUBE subclause.
2. Execute the modified query and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab14\74 - Lab Exercise 3 - Task 3 Result.txt. Notice the number of rows in your results.
3. What is the difference between the ROLLUP and CUBE subclauses?
4. Which is the more appropriate subclause to use in this example?

▶ Task 4: Analyze the Total Sales Value by Year and Month

1. Write a SELECT statement against the Sales.OrderValues view and retrieve these columns:
   - Calculated column with the alias groupid (use the GROUPING_ID function with the order year and order month as the input parameters).
   - Year of the orderdate column as orderyear.
   - Month of the orderdate column as ordermonth.
   - Total sales value using the val column as salesvalue.

- Since year and month form a hierarchy, return all interesting grouping sets based on the orderyear and ordermonth columns and sort the result by groupid, orderyear, and ordermonth.

2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab14\75 - Lab Exercise 3 - Task 4 Result.txt.

Results: After this exercise, you should have an understanding of how to use the GROUPING SETS, CUBE, and ROLLUP subclauses in T-SQL statements.

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways

Review Question(s)

Question: Once a dataset has been pivoted with aggregation, can the original detail rows be restored with an unpivot operation?

Question: What are the possible sources of NULLs returned by a query using grouping sets to create aggregations?

Question: Which subclause infers a hierarchy of columns to create meaningful grouping sets?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 15

Executing Stored Procedures

Contents:

Module Overview	**15-1**
Lesson 1: Querying Data with Stored Procedures	**15-2**
Lesson 2: Passing Parameters to Stored Procedures	**15-5**
Lesson 3: Creating Simple Stored Procedures	**15-9**
Lesson 4: Working with Dynamic SQL	**15-12**
Lab: Executing Stored Procedures	**15-16**
Module Review and Takeaways	**15-22**

Module Overview

In addition to writing stand-alone SELECT statements to return data from Microsoft® SQL Server®, you may need to execute T-SQL procedures created by an administrator or developer and stored in a database. This module will show you how to execute stored procedures, including how to pass parameters into procedures written to accept them. This module will also show you how basic stored procedures are created, providing a better understanding of what happens on the server when you execute one. Finally, this module will show you how to generate dynamic SQL statements, which is often a requirement in development environments where stored procedures are not being used.

Objectives

After completing this module, you will be able to:

- Return results by executing stored procedures.
- Pass parameters to procedures.
- Create simple stored procedures that encapsulate a SELECT statement.
- Construct and execute dynamic SQL with EXEC and sp_executesql.

Lesson 1
Querying Data with Stored Procedures

Many reporting and development tools offer the choice between writing and executing specific T-SQL SELECT statements, and choosing from queries saved as stored procedures in SQL Server. While stored procedures can encapsulate most T-SQL operations, including system administration tasks, this lesson will focus on using stored procedures to return results sets, as an alternative to writing your own SELECT statements.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe stored procedures and their use.
- Write T-SQL statements that execute stored procedures to return data.

Examining Stored Procedures

Stored procedures are named collections of T-SQL statements created with the CREATE PROCEDURE command. They encapsulate many server and database commands, and can provide a consistent application programming interface (API) to client applications using input parameters, output parameters, and return values.

- Stored procedures are collections of T-SQL statements stored in a database
- Procedures can return results, manipulate data, and perform administrative actions on the server
- With other objects, procedures can provide a trusted application programming interface to a database, insulating applications from database structure changes
  - Use views, functions, and procedures to return data
  - Use procedures to modify and add new data
  - Alter procedure definition in one place, rather than update application code

Since this course focuses primarily on retrieving results from databases through SELECT statements, this lesson will only cover the use of stored procedures that encapsulate SELECT queries. However, it may be useful to note that stored procedures can also include INSERT, UPDATE, DELETE, and other valid T-SQL commands. In addition, they can be used to provide an interface layer between a database and an application. Using such a layer, developers and administrators can ensure that all activity is performed by trusted code modules that validate input and handle errors appropriately. Elements of such an API would include:

- Views or table-valued functions as wrappers for simple retrieval.
- Stored procedures for retrieval when complex validation or manipulation is required.
- Stored procedures for inserting, updating, or deleting rows.

In addition to encapsulating code and making it easier to maintain, this approach provides a security layer. Users may be granted access to objects rather than the underlying tables themselves. This ensures that users may only use the provided application to access data rather than other tools.

Stored procedures offer other benefits as well, including network and database engine performance improvements. See Microsoft course 20464D: *Developing Microsoft SQL Server Databases* for additional information on these benefits and more details on creating and using stored procedures.

Go to Books Online at:

Stored Procedures (Database Engine)

http://go.microsoft.com/fwlink/?LinkID=394809

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

Executing Stored Procedures

Earlier in this course, you learned how to execute system stored procedures. The same mechanism exists for executing user procedures. Therefore some of the following guidelines are provided for review:

- Invoke a stored procedure using EXECUTE or EXEC
- Call procedure with two-part name
- Pass parameters in @name=value form, using appropriate data type

```
EXEC Production.ProductsbySuppliers
    @supplierid = 1;

EXEC Production.ProductsbySuppliers
    @supplierid = 1, @numrows = 2;
```

- To execute a stored procedure, use the EXECUTE command or its shortcut, EXEC, followed by the two-part name of the procedure. Your reporting tool may provide a graphical interface for selecting procedures by name, which will invoke the EXEC command for you.
- If the procedure accepts parameters, pass them as name-value pairs. For example, if the parameter is called custid and the value to pass is 5, use this form: @custid=5. Multiple parameters are separated with commas.
- Pass parameters of the appropriate data type to the stored procedure. For example, if a procedure accepts an NVARCHAR, pass in the Unicode character string format: N'string'.
- If the procedure encapsulates a simple SELECT statement, no additional elements are needed to execute it. If the procedure includes an OUTPUT parameter, additional steps will be required. See the lesson on OUTPUT parameters later in this module.

Note: You may see sample code that omits the use of the EXEC command before the name of a procedure. While this works on the first line of a batch (or in the only line of a one-line batch), this is not a best practice. Always use EXECUTE or EXEC to invoke stored procedures.

Go to Books Online at:

Execute a Stored Procedure

http://go.microsoft.com/fwlink/?LinkID=402790

Demonstration: Querying Data with Stored Procedures

In this demonstration, you will see how to:

- Use stored procedures.

Demonstration Steps

Use Stored Procedures

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod15\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod15\Demo folder.

5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

Question: You have the following query, which is intended to call a stored procedure called HumanResources.FilteredSkills:

```
EXEC HumanResources.FilteredSkills
   departmentid = @1, skilllevel = @4;
```

Your query returns an error. What should you do to resolve the error?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Passing Parameters to Stored Procedures

Procedures can be written to accept parameters to provide greater flexibility. Most parameters are written as input parameters, which accept values passed in the EXEC statement and are used inside the procedure. Some procedures may also return values in the form of OUTPUT parameters, which require additional handling by the client when invoking the procedure. You will learn how to pass input and return output parameters in this lesson.

Lesson Objectives

After completing this lesson, you will be able to:

- Write EXECUTE statements that pass input parameters to stored procedures.
- Write T-SQL batches that prepare output parameters and execute stored procedures.

Passing Input Parameters to Stored Procedures

Stored procedures can be written to accept input parameters to provide greater flexibility. Procedures declare their parameters by name and data type in the header of the CREATE PROCEDURE statement, and then use the parameters as local variables in the body of the procedure. For example, an input parameter might be used in the predicate of a WHERE clause or as the value in a TOP operator.

- Parameters are defined in the header of the procedure code, including name, data type and direction (input is default)
- Parameters are discoverable using SQL Server Management Studio and the sys.parameters view
- To pass parameters in an EXEC statement, use names defined in procedure

```
CREATE PROCEDURE Production.ProductsbySuppliers
(@supplierid AS INT)
AS ...
```

```
EXEC Production.ProductsbySuppliers
    @supplierid = 1;
```

To call a stored procedure and pass parameters, use the following syntax:

Stored Procedure with Parameters Syntax

```
EXEC <schema_name>.<procedure_name> @<parameter_name> = <VALUE> [, ...]
```

For example, if you have a procedure called ProductsBySuppliers stored in the Production schema and it accepts a parameter named supplierid, you would use the following:

Stored Procedure with Parameters Example

```
EXEC Production.ProductsBySuppliers @supplierid = 1;
```

To pass multiple input parameters, separate the name-value pairs with commas, as in this example:

Stored Procedure with Multiple Parameters Example

```
EXEC Sales.FindOrder @empid = 1, @custid=1;
```

Note: The previous example refers to a procedure that does not exist in the sample database for the course. Other examples in the demonstration script for this lesson can be executed against procedures in the sample TSQL database.

If you have not been provided with the names and data types of the parameters for the procedures you will be executing, you can typically discover them yourself, assuming you have permissions to do so. SQL Server Management Studio (SSMS) displays a parameters folder below each stored procedure, which lists the names, types, and direction (input/output) of each defined parameter. Alternatively, you can query a system catalog view such as sys.parameters to retrieve parameter definitions. For an example, see the demonstration script provided for this lesson.

Additional reading about passing parameters to stored procedures can be found in Books Online at:

Specify Parameters

http://go.microsoft.com/fwlink/?LinkID=402792

Working with OUTPUT Parameters

So far in this module, you have seen procedures that return results through an embedded SELECT statement. SQL Server also provides the capability to return a scalar value through a parameter marked as an OUTPUT parameter. This has several benefits: A procedure can return a result set via a SELECT statement and provide an additional value, such as a row count, to the calling application. For some specific scenarios where only a single value is desired, a procedure that returns an OUTPUT parameter can perform faster than a procedure that returns the scalar value in a result set.

- Output parameters allow you to return values from a stored procedure
  - Compare with returning a result set
- Parameter marked for output in procedure header and in calling query

```
CREATE PROCEDURE <proc_name>
(@<input_param> AS <type>,
 @<output_param> AS <type> OUTPUT)
 AS ...
```

```
DECLARE @<output_param> AS <type>;
 EXEC <proc_name> <input_parameter_list>,
@<output_param> OUTPUT;
 SELECT @output_param;
```

There are two aspects to working with stored procedures using output parameters:

1. The procedure itself must mark a parameter with the OUTPUT keyword in the parameter declaration.

See the following example:

Creating a Stored Procedure with an OUTPUT Parameter Example

```
CREATE PROCEDURE Sales.GetCustPhone
(@custid AS INT, @phone AS nvarchar(24) OUTPUT)
AS ...
```

2. The T-SQL batch that calls the procedure must provide additional code to handle the output parameter. The code includes a local variable that acts as a container for the value that will be returned by the procedure when it executes. The parameter is added to the EXEC statement, marked with the OUTPUT keyword. After the stored procedure has completed, the variable will contain the value of the output parameter set inside the procedure.

The following example declares a local variable to be passed as the output parameter, executes a procedure, and then examines the variable with a SELECT statement:

Executing a Stored Procedure with OUTPUT Parameter Example

```
DECLARE @customerid INT =5, @phonenum NVARCHAR(24);
EXEC Sales.GetCustPhone @custid=@customerid, @phone=@phonenum OUTPUT;
SELECT @phonenum AS phone;
```

MCT USE ONLY STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Passing Parameters to Stored Procedures

In this demonstration, you will see how to:

- Pass parameters to a stored procedure.

Demonstration Steps

Pass Parameters to a Stored Procedure

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod15\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod15\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Question

A DBA has created a stored procedure in the HumanResources database by executing the following:
 	
`CREATE PROCEDURE HumanResources.SkillsForEmployee (@empid AS INT)`
`AS`
`  SELECT e.ID, e.FirstName, e.LastName, s.SkillName, s.Level`
`  FROM HumanResources.Employees AS e`
`  JOIN HumanResources.Skills AS s ON e.ID = s.EmployeeID`
`  WHERE e.ID = @empid`
`GO`
 	
You call the procedure with the following statement:
 	
`EXEC HumanResources.SkillsForEmployee @empid = N'24'`
 	
Your query returns an error. What should you do to fix your query?

Select the correct answer.	
	Pass the @empid parameter as an integer instead of an nvarchar.
	Move the position of the "@" symbol to the correct place.
	Add a value for the e.ID parameter to your query.
	Instead of using the stored procedure, execute your own SELECT query.
	Add the OUTPUT keyword to @empid parameter.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Creating Simple Stored Procedures

To better understand how to work with stored procedures written by developers and administrators, it is useful to learn how they are created. In this lesson, you will see how to write a stored procedure that returns a result set from an encapsulated SELECT statement.

Lesson Objectives

After completing this lesson, you will be able to:

- Use the CREATE PROCEDURE statement to write a stored procedure.
- Create a stored procedure that accepts input parameters.

Creating Procedures to Return Rows

Stored procedures in SQL Server are used for many tasks, including system configuration and maintenance, as well as data manipulation. As previously mentioned, there are advantages to creating procedures to standardize access to data. To do that, you can create a stored procedure that is a wrapper for a SELECT statement, which may include any of the data manipulations you have already learned in this course.

- Stored procedures can be wrappers for simple or complex SELECT statements
- Procedures may include input and output parameters as well as return values
- Use CREATE PROCEDURE statement:

```
CREATE PROCEDURE <schema_name.proc_name>
(<parameter_list)
 AS
SELECT <body of SELECT statement>;
```

- Modify design of procedure with ALTER PROCEDURE statement
  - No need to drop, recreate

The following example creates a procedure that aggregates order information:

Example of a Procedure That Returns Rows

```
CREATE PROCEDURE Sales.OrderSummaries
AS
SELECT O.orderid, O.custid, O.empid, O.shipperid, CAST(O.orderdate AS date)AS orderdate,
  SUM(OD.qty) AS quantity,
  CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount))
       AS NUMERIC(12, 2)) AS ordervalue
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
GROUP BY O.orderid, O.custid, O.empid, O.shipperid, O.orderdate;
GO
```

To execute this procedure, use the EXECUTE or EXEC command before the procedure's two-part name:

Executing a Procedure That Returns Rows

```
EXEC [Sales].[OrderSummaries];
```

A partial result:

```
orderid custid empid shipperid orderdate  quantity ordervalue
------- ----- ------ --------- ---------- -------- ----------
10248       85      5         3 2006-07-04       27     440.00
10249       79      6         1 2006-07-05       49    1863.40
10250       34      4         2 2006-07-08       60    1552.60
```

To modify the design of the procedure, such as to change the columns in the SELECT list or add an ORDER BY clause, use the ALTER PROCEDURE (abbreviated ALTER PROC) statement and supply the full new code for the procedure.

See the following example:

Altering a Stored Procedure That Returns Rows

```
ALTER PROCEDURE Sales.OrderSummaries
AS
SELECT O.orderid, O.custid, O.empid, O.shipperid, CAST(O.orderdate AS date)AS orderdate,
  SUM(OD.qty) AS quantity,
  CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount))
       AS NUMERIC(12, 2)) AS ordervalue
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
GROUP BY O.orderid, O.custid, O.empid, O.shipperid, O.orderdate
ORDER BY orderid, orderdate;
```

Changing the procedure with ALTER PROCEDURE is preferable to using DROP PROCEDURE to delete it, and then using CREATE PROCEDURE to rebuild it with a new definition. By altering it in place, security permissions do not need to be reassigned.

For more information on modifying stored procedures, go to Books Online at:

Modify a Stored Procedure

http://go.microsoft.com/fwlink/?LinkID=402793

Creating Procedures That Accept Parameters

A stored procedure that accepts input parameters provides added flexibility to its use. To define input parameters in your own stored procedures, declare them in the header of the CREATE PROCEDURE statement, then refer to them in the body of the stored procedure. Define the parameters with an @ prefix in the name, then assign them a data type.

- Input parameters passed to procedure logically behave like local variables within procedure code
- Assign name with @prefix, data type in procedure header
- Refer to parameter in body of procedure

```
CREATE PROCEDURE Production.ProdsByCategory
(@numrows AS int, @catid AS int)
 AS
SELECT TOP(@numrows) productid,
        productname, unitprice
FROM   Production.Products
WHERE          categoryid = @catid;
```

Note: Parameters may also be assigned default values, including NULL.

See the following example:

Syntax of a Stored Procedure That Accepts Parameters

```
CREATE PROCEDURE <schema>.<procedure_name>
(@<parameter_name> AS <data_type>)
AS ...
```

MCT USE ONLY, STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

For example, the following procedure will accept the empid parameter as an integer and pass it to the WHERE clause to be used as a filter:

Example of a Stored Procedure That Accepts Parameters

```
CREATE PROCEDURE Sales.OrderSummariesByEmployee
(@empid AS int)
AS
SELECT O.orderid, O.custid, O.empid, O.shipperid, CAST(O.orderdate AS date)AS orderdate,
  SUM(OD.qty) AS quantity,
  CAST(SUM(OD.qty * OD.unitprice * (1 - OD.discount))
       AS NUMERIC(12, 2)) AS ordervalue
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
WHERE empid = @empid
GROUP BY O.orderid, O.custid, O.empid, O.shipperid, O.orderdate
ORDER BY orderid, orderdate;
GO
```

To call the procedure, use EXEC and pass in a value:

Executing a Stored Procedure That Accepts Parameters

```
EXEC Sales.OrderSummariesByEmployee @empid = 5;
```

Demonstration: Creating Simple Stored Procedures

In this demonstration, you will see how to:

- Create a stored procedure.

Demonstration Steps

Create a Stored Procedure

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod15\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod15\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

   Question: The HumanResources.SkillLevelsForDepartment stored procedure is a popular procedure that ensures skills data can be examined in an anonymous form. You have been asked to add a new parameter to the stored procedure. Why should you use ALTER PROCEDURE instead of DROP PROCEDURE followed by CREATE PROCEDURE.

Lesson 4
Working with Dynamic SQL

In organizations where creating parameterized stored procedures is not supported, you may need to execute T-SQL code constructed in your application at runtime. Dynamic SQL provides a mechanism for constructing a character string that is passed to SQL Server, interpreted as a command, and executed.

In this lesson, you will learn how to pass dynamic SQL queries to SQL Server, using the EXEC statement and the system procedure sp_executesql.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how T-SQL can be dynamically constructed.
- Write queries that use dynamic SQL.

Constructing Dynamic SQL

Dynamic SQL provides a mechanism for constructing a character string that is passed to SQL Server, interpreted as a command, and executed. Why would you want to do this? You may not know all the values necessary for your query until execution time, such as taking the results of one query and using them as inputs to another (for example, a pivot query) or an administrative maintenance routine that accepts object names at runtime.

- Dynamic SQL is T-SQL code assembled into a character string, interpreted as a command, and executed
- Dynamic SQL provides flexibility for administrative and programming tasks
- Two methods for dynamically executing SQL statements:
  - EXEC command can accept a string as input in parentheses. No parameters may be passed in
  - System-stored procedure sp_executesql (preferred) supports parameters
- Beware of risks from unvalidated inputs in dynamic SQL!

T-SQL supports two methods for building dynamic SQL expressions—using the EXECUTE command (or its shortcut EXEC) with a string or invoking the system-stored procedure sp_executesql:

1. The EXECUTE or EXEC command supports the use of a string as an input in the following form, but does not support parameters, which need to be combined in the input string:

The following example shows how individual strings may be concatenated to form a command:

Dynamic SQL Example

```
DECLARE @sqlstring AS VARCHAR(1000);
SET @sqlstring='SELECT empid,' + ' lastname '+' FROM HR.employees;'
EXEC(@sqlstring);
GO
```

2. The system-stored procedure sp_executesql supports string input for the query, as well as input parameters.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The following example shows a simple string with a parameter passed to sp_executesql:

Passing Dynamic SQL With sp_executesql

```
DECLARE @sqlcode AS NVARCHAR(256) = N'SELECT GETDATE() AS dt';
EXEC sys.sp_executesql @statement = @sqlcode;
GO
```

It is important to know that EXEC cannot accept parameters and does not promote query plan reuse. Therefore, it is preferred that you use sp_executesql for passing dynamic SQL to SQL Server.

Go to the section *Using EXECUTE with a Character String* in the *EXECUTE (Transact-SQL)* topic in Books Online at:

EXECUTE (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402794

For more information on using sp_executesql, see the next topic in this lesson.

Writing Queries with Dynamic SQL

In the previous topic, you learned that there were two methods for executing dynamic SQL. This topic focuses on the preferred method, calling sp_executesql.

Constructing and executing dynamic SQL with sp_executesql is preferred over using EXEC because EXEC cannot take parameters at runtime. In addition, sp_executesql generates execution plans that are more likely to be reused than EXEC. Perhaps most important, though, using sp_executesql can provide a line of defense against SQL injection attacks by defining data types for parameters.

To use sp_executesql, provide a character string value that contains the query code as a parameter, as in the following syntax example:

sp_executesql Syntax Example

```
DECLARE @sqlcode AS NVARCHAR(256) = N'<code_to_run>';
EXEC sys.sp_executesql @statement = @sqlcode;
GO
```

The following example uses sp_executesql to execute a simple SELECT query:

sp_executesql Example

```
DECLARE @sqlcode AS NVARCHAR(256) =
      N'SELECT GETDATE() AS dt';
EXEC sys.sp_executesql @statement = @sqlcode;
GO
```

MCT USE ONLY STUDENT USE PROHIBITED

To use sp_executesql with parameters, provide the query code, as well as two additional parameters:

- @stmt, a Unicode string variable to hold the query text
- @params, a Unicode string variable that holds a comma-separated list of parameter names and data types

In addition to these two variables, you will declare and assign variables to hold the values for the parameters you wish to pass in to sp_executesql.

The following example uses sp_executesql to dynamically generate a query that returns an employee's information based on an empid value:

Using sp_executesql With Parameters

```
DECLARE @sqlstring AS NVARCHAR(1000);
DECLARE @empid AS INT;
SET @sqlstring=N'SELECT empid, lastname FROM HR.employees WHERE empid=@empid;'
EXEC sys.sp_executesql @statement = @sqlstring, @params=N'@empid AS INT',
      @empid = 5;
```

The result:

```
empid lastname
----- --------
5     Buck
```

Note: sp_executesql can also use output parameters marked with the OUTPUT keyword, which you learned about earlier in this module.

For a discussion about query plan reuse and more coverage of sp_executesql, go to Books Online at:

Using sp_executesql

http://go.microsoft.com/fwlink/?LinkID=402795

Demonstration: Working with Dynamic SQL

In this demonstration, you will see how to:

- Execute dynamic SQL queries.

Demonstration Steps

Execute Dynamic SQL Queries

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod15\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod15\Demo folder.
3. In Solution Explorer, open the **41 – Demonstration D.sql** script file.

4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

Question: You want to execute dynamic SQL with a single parameter named @skillname. In addition to the parameter itself, what other parameters should you send to sp_executesql?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Executing Stored Procedures

Scenario

You are a business analyst for Adventure Works, who will be writing reports using corporate databases stored in SQL Server 2014. You have been provided with a set of business requirements for data and will write T-SQL queries to retrieve the specified data from the databases. You have learned that some of the data can only be accessed via stored procedures instead of directly querying the tables. Additionally, some of the procedures require parameters in order to interact with them.

Objectives

After completing this lab, you will be able to:

- Use the EXECUTE statement to invoke stored procedures.
- Pass parameters to stored procedures.
- Execute system stored procedures.

Estimated Time: 30 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Using the EXECUTE Statement to Invoke Stored Procedures

Scenario

The IT department has supplied T-SQL code to create a stored procedure to retrieve the top 10 customers by the total sales amount. You will practice how to execute a stored procedure.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Create and Execute a Stored Procedure
3. Modify the Stored Procedure and Execute It

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab15\Starter folder as Administrator.

▶ Task 2: Create and Execute a Stored Procedure

1. In SQL Server Management Studio, open the project file D:\Labfiles\Lab15\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.
2. Execute the provided T-SQL code to create the stored procedure Sales.GetTopCustomers:

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

3. CREATE PROCEDURE Sales.GetTopCustomers AS

```
SELECT TOP(10)
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC;
```

4. Write a T-SQL statement to execute the created procedure.
5. Execute the T-SQL statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab15\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.
6. What is the difference between the previous T-SQL code and this one?
7. If some applications are using the stored procedure from task 1, would they still work properly after the changes you have applied in task 2?

▶ Task 3: Modify the Stored Procedure and Execute It

1. The IT department has changed the stored procedure from task 1 and supplied you with T-SQL code to apply the needed changes. Execute the provided T-SQL code:

```
ALTER PROCEDURE Sales.GetTopCustomers AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

2. Write a T-SQL statement to execute the modified stored procedure.
3. Execute the T-SQL statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab15\Solution\53 - Lab Exercise 1 - Task 2 Result.txt.

Results: After this exercise, you should be able to invoke a stored procedure using the EXECUTE statement.

Exercise 2: Passing Parameters to Stored Procedures

Scenario

The IT department supplied you with additional modifications of the stored procedure in task 1. The modified stored procedure lets you pass parameters that specify the order year and number of customers to retrieve. You will practice how to execute the stored procedure with a parameter.

The main tasks for this exercise are as follows:

1. Execute a Stored Procedure with a Parameter for Order Year
2. Modify the Stored Procedure to have a Default Value for the Parameter
3. Pass Multiple Parameters to the Stored Procedure

4. Return the Result from a Stored Procedure Using the OUTPUT Clause

▶ Task 1: Execute a Stored Procedure with a Parameter for Order Year

1. Open the SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. Execute the provided T-SQL code to modify the Sales.GetTopCustomers stored procedure to include a parameter for order year (@orderyear):

```
ALTER PROCEDURE Sales.GetTopCustomers
@orderyear int
AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE YEAR(o.orderdate) = @orderyear
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

3. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure for the year 2007.
4. Execute the T-SQL statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab15\Solution\62 - Lab Exercise 2 - Task 1_1 Result.txt.
5. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure for the year 2008.
6. Execute the T-SQL statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab15\Solution\63 - Lab Exercise 2 - Task 1_2 Result.txt.
7. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure without a parameter.
8. Execute the T-SQL statement. What happened? What is the error message?
9. If an application was designed to use the exercise 1 version of the stored procedure, would the modification made to the stored procedure in this exercise impact the usability of that application? Please explain.

▶ Task 2: Modify the Stored Procedure to have a Default Value for the Parameter

1. Execute the provided T-SQL code to modify the Sales.GetTopCustomers stored procedure:

```
ALTER PROCEDURE Sales.GetTopCustomers
@orderyear int = NULL
AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE YEAR(o.orderdate) = @orderyear OR @orderyear IS NULL
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

2. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure without a parameter.
3. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\64 - Lab Exercise 2 - Task 2 Result.txt.
4. If an application was designed to use the Exercise 1 version of the stored procedure, would the change made to the stored procedure in this task impact the usability of that application? How does this change influence the design of future applications?

▶ Task 3: Pass Multiple Parameters to the Stored Procedure

1. Execute the provided T-SQL code to add the parameter @n to the Sales.GetTopCustomers stored procedure. You use this parameter to specify how many customers you want retrieved. The default value is 10.

```
ALTER PROCEDURE Sales.GetTopCustomers
@orderyear int = NULL,
@n int = 10
AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE YEAR(o.orderdate) = @orderyear OR @orderyear IS NULL
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT @n ROWS ONLY;
```

2. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure without any parameters.
3. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\65 - Lab Exercise 2 - Task 3_1 Result.txt.
4. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure for order year 2008 and five customers.
5. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\66 - Lab Exercise 2 - Task 3_2 Result.txt.
6. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure for the order year 2007.
7. Execute the T-SQL statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab15\Solution\67 - Lab Exercise 2 - Task 3_3 Result.txt.
8. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure to retrieve 20 customers.
9. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\68 - Lab Exercise 2 - Task 3_4 Result.txt.
10. Do the applications using the stored procedure need to be changed because another parameter was added?

MCT USE ONLY STUDENT USE PROHIBITED

▶ Task 4: Return the Result from a Stored Procedure Using the OUTPUT Clause

1. Execute the provided T-SQL code to modify the Sales.GetTopCustomers stored procedure to return the customer contact name based on a specified position in a ranking of total sales, which is provided by the parameter @customerpos. The procedure also includes a new parameter named @customername, which has an OUTPUT option:

```
ALTER PROCEDURE Sales.GetTopCustomers
@customerpos int = 1,
@customername nvarchar(30) OUTPUT
AS
SET @customername = (
SELECT
c.contactname
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
GROUP BY c.custid, c.contactname
ORDER BY SUM(o.val) DESC
OFFSET @customerpos - 1 ROWS FETCH NEXT 1 ROW ONLY
);
```

2. The IT department also supplied you with T-SQL code to declare the new variable @outcustomername. You will use this variable as an output parameter for the stored procedure.
3. DECLARE @outcustomername nvarchar(30);
4. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure and retrieve the first customer.
5. Write a SELECT statement to retrieve the value of the output parameter @outcustomername.
6. Execute the batch of T-SQL code consisting of the provided DECLARE statement, the written EXECUTE statement, and the written SELECT statement.
7. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\69 - Lab Exercise 2 - Task 4 Result.txt.

Results: After this exercise, you should know how to invoke stored procedures that have parameters.

Exercise 3: Executing System Stored Procedures

Scenario

In the previous module, you learned how to query the system catalog. Now you will practice how to execute some of the most commonly used system-stored procedures to retrieve information about tables and columns.

The main tasks for this exercise are as follows:

1. Execute the Stored Procedure sys.sp_help
2. Execute the Stored Procedure sys.sp_helptext
3. Execute the Stored Procedure sys.sp_columns
4. Drop the Created Stored Procedure

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 1: Execute the Stored Procedure sys.sp_help

1. Open the SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. Write an EXECUTE statement to invoke the sys.sp_help stored procedure without a parameter.
3. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\72 - Lab Exercise 3 - Task 1_1 Result.txt.
4. Write an EXECUTE statement to invoke the sys.sp_help stored procedure for a specific table by passing the parameter Sales.Customers.
5. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\73 - Lab Exercise 3 - Task 1_2 Result.txt.

▶ Task 2: Execute the Stored Procedure sys.sp_helptext

1. Write an EXECUTE statement to invoke the sys.sp_helptext stored procedure, passing the Sales.GetTopCustomers stored procedure as a parameter.
2. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\74 - Lab Exercise 3 - Task 2 Result.txt.

▶ Task 3: Execute the Stored Procedure sys.sp_columns

1. Write an EXECUTE statement to invoke the sys.sp_columns stored procedure for the table Sales.Customers. You will have to pass two parameters: @table_name and @table_owner.
2. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\75 - Lab Exercise 3 - Task 3 Result.txt.

▶ Task 4: Drop the Created Stored Procedure

1. Execute the provided T-SQL statement to remove the Sales.GetTopCustomers stored procedure:

```
DROP PROCEDURE Sales.GetTopCustomers;
```

Results: After this exercise, you should have a basic knowledge of invoking different system-stored procedures.

Module Review and Takeaways

Review Question(s)

Question: What benefits do stored procedures provide for data retrieval that views do not?

Question: What form should parameter and value pairs take when passed to a stored procedure in the EXECUTE statement?

Question: Which method for constructing dynamic SQL allows parameters to be passed at runtime?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 16

Programming with T-SQL

Contents:

Module Overview	**16-1**
Lesson 1: T-SQL Programming Elements	**16-2**
Lesson 2: Controlling Program Flow	**16-7**
Lab: Programming with T-SQL	**16-11**
Module Review and Takeaways	**16-17**

Module Overview

In addition to the data retrieval and manipulation statements you have learned about in this course,

T-SQL provides some basic programming features, such as variables, control-of-flow elements, and conditional execution. In this module, you will learn how to enhance your T-SQL code with programming elements.

Objectives

After completing this module, you will be able to:

- Describe the language elements of T-SQL used for simple programming tasks.
- Describe batches and how they are handled by SQL Server.
- Declare and assign variables and synonyms.
- Use IF and WHILE blocks to control program flow.

Lesson 1
T-SQL Programming Elements

With a few exceptions, most of your work with T-SQL in this course so far has focused on single-statement structures, such as SELECT statements. As you move from executing code objects to creating them, you will need to understand how multiple statements interact with the server on execution. You will also need to be able to temporarily store values. For example, you might need to temporarily store values that will be used as parameters in stored procedures. Finally, you may want to create aliases, or pointers, to objects so that you can reference them by a different name or from a different location than where they are defined. This lesson will cover each of these topics.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how Microsoft® SQL Server® treats collections of statements as batches.
- Create and submit batches of T-SQL code for execution by SQL Server.
- Describe how SQL Server stores temporary objects as variables.
- Write code that declares and assigns variables.
- Create and invoke synonyms.

Introducing T-SQL Batches

T-SQL batches are collections of one or more T-SQL statements that are submitted to SQL Server by a client as a single unit. SQL Server operates on all the statements in a batch at the same time when parsing, optimizing, and executing the code.

If you are a report writer tasked primarily with writing SELECT statements and not procedures, it is still important to understand batch boundaries, since they will affect your work with variables and parameters in stored procedures and other routines. As you will see, you must declare a variable in the same batch in which it is referenced. It is important, therefore, to recognize what is contained in a batch.

Batches are delimited by the client application, and how you mark the end of a batch will depend on the settings of your client. For example, the default batch terminator in SQL Server Management Studio (SSMS) is the keyword GO. This is not a T-SQL keyword, but is one recognized by SSMS to indicate the end of a batch.

When working with T-SQL batches, there are two important considerations to keep in mind:

- Batches are boundaries for variable scope, which means that a variable defined in one batch may only be referenced by other code in the same batch.
- Some statements, typically data definition statements such as CREATE VIEW, may not be combined with others in the same batch. See Books Online for the complete list.

MCT USE ONLY. STUDENT USE PROHIBITED

Additional reading can be found in Books Online at:

Batches

http://go.microsoft.com/fwlink/?LinkID=402796

Working with Batches

As you have seen, batches are collections of T-SQL statements submitted as a unit to SQL Server for parsing, optimization, and execution. Understanding how batches are parsed will be useful in identifying error messages and behavior.

- Batches are parsed for syntax as a unit
  - Syntax errors cause the entire batch to be rejected
  - Runtime errors may allow the batch to continue after failure, by default

```
--Valid batch
INSERT INTO dbo.t1 VALUES(1,2,N'abc');
INSERT INTO dbo.t1 VALUES(2,3,N'def');
GO
--invalid batch
INSERT INTO dbo.t1 VALUE(1,2,N'abc');
INSERT INTO dbo.t1 VALUES(2,3,N'def');
GO
```

- Batches can contain error-handling code

When a batch is submitted by a client (such as when you press the Execute button in SSMS), the batch is parsed for syntax errors by the SQL Server engine. Any errors found will cause the entire batch to be rejected; there will be no partial execution of statements within the batch.

If the batch passes the syntax check, then SQL Server proceeds with additional steps—resolving object names, checking permissions, and optimizing the code for execution. Once this process completes and execution begins, statements succeed or fail individually. This is an important contrast to syntax checking. If a runtime error occurs on one line, the next line may be executed, unless you've added error handling to the code.

Note: Error handling will be covered in a later module.

For example, the following batch contains a syntax error in the first line:

Batch With Error

```
INSERT INTO dbo.t1 VALUE(1,2,N'abc');
INSERT INTO dbo.t1 VALUES(2,3,N'def');
GO
```

Upon submitting the batch, the following error is returned:

```
Msg 102, Level 15, State 1, Line 1
Incorrect syntax near 'VALUE'.
```

The error occurred in line 1, but the entire batch is rejected, and execution does not continue with line 2. Even if the lines were reversed and the syntax error occurred in the second line, the first line would not be executed since the entire batch would be rejected.

MCT USE ONLY STUDENT USE PROHIBITED

Introducing T-SQL Variables

In T-SQL, as with other programming languages, variables are objects that allow temporary storage of a value for later use. You have already encountered variables in this course, using them to pass parameter values to stored procedures and functions.

In T-SQL, variables must be declared before they can be used. They may be assigned a value, or initialized, when they are declared. Declaring a variable includes providing a name and a data type, as shown below.

As you have previously learned, variables must be declared in the same batch in which they are referenced. In other words, all T-SQL variables are local in scope to the batch, both in visibility and lifetime. Only other statements in the same batch can see a variable declared in the batch. A variable is automatically destroyed when the batch ends.

The following example shows the use of variables to store values that will be passed to a stored procedure in the same batch:

Using Variables

```
--Declare and initialize the variables.
DECLARE @numrows INT = 3, @catid INT = 2;
--Use variables to pass the parameters to the procedure.
EXEC Production.ProdsByCategory
      @numrows = @numrows, @catid = @catid;
GO
```

Additional reading can be found in Books Online at:

Variables (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402797

Working with Variables

Once you have declared a variable, you must initialize it, or assign it a value. You may do that in three ways:

- In SQL Server 2008 or later, you may initialize a variable using the DECLARE statement.
- In any version of SQL Server, you may assign a single (scalar) value using the SET statement.
- In any version of SQL Server, you can assign a value to a variable using a SELECT statement. Be sure that the SELECT statement returns exactly one row. An empty result will leave the variable with its original value; more than one result will cause an error.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The following example shows the three ways of declaring and assigning values to variables:

Declaring and Assigning Values to Variables

```
DECLARE @var1 AS INT = 99;
DECLARE @var2 AS NVARCHAR(255);
SET @var2 = N'string';
DECLARE @var3 AS NVARCHAR(20);
SELECT @var3 = lastname FROM HR.Employees WHERE empid=1;
SELECT @var1 AS var1, @var2 AS var2, @var3 AS var3;
GO
```

The results are:

```
var1 var2   var3
---- ------ ----
99   string Davis
```

Working with Synonyms

In SQL Server, synonyms provide a method for creating a link, or alias, to an object stored in the same database or even on another instance of SQL Server. Objects that may have synonyms defined for them include tables, views, stored procedures, and user-defined functions.

- A synonym is an alias or link to an object stored either on the same SQL Server instance or on a linked server
  - Synonyms can point to tables, views, procedures, and functions
- Synonyms can be used for referencing remote objects as though they were located locally, or for providing alternative names to other local objects
- Use the CREATE and DROP commands to manage synonyms

```
USE tempdb;
GO
CREATE SYNONYM dbo.ProdsByCategory FOR
        TSQL.Production.ProdsByCategory;
GO
EXEC dbo.ProdsByCategory
        @numrows = 3, @catid = 2;
```

Synonyms can be used to make a remote object appear local or to provide an alternative name for a local object. For example, synonyms can be used to provide an abstraction layer between client code and the actual database objects used by the code. The code references objects by their aliases, regardless of the object's actual name.

Note: You can create a synonym which points to an object that does not yet exist. This is called deferred name resolution. The SQL Server engine will not check for the existence of the actual object until the synonym is used at runtime.

To manage synonyms, use the DDL commands CREATE SYNONYM, ALTER SYNONYM, and DROP SYNONYM, as in the following example:

Managing Synonyms

```
CREATE SYNONYM dbo.ProdsByCategory FOR TSQL.Production.ProdsByCategory;
GO
EXEC dbo.ProdsByCategory @numrows = 3, @catid = 2;
```

To create a synonym, you must have CREATE SYNONYM permission as well as permission to alter the schema in which the synonym will be stored.

Go to *Using Synonyms (Database Engine)* in Books Online at:

Using Synonyms (Database Engine)

http://go.microsoft.com/fwlink/?LinkID=402798

Demonstration: T-SQL Programming Elements

In this demonstration, you will see how to:

- Control batch execution and variable usage.

Demonstration Steps

Control Batch Execution and Variable Usage

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod16\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod16\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

   Question: You have the following T-SQL script:

```
INSERT INTO HumanResources.PossibleSkills (SkillName, Category, Credit)
  VALUES('Database Administration', 'IT Professional', 5);
INSERT INTO HumanResources.PossibleSkills (SkillName, Category, Credit)
  VALUES('C#.NET', 'Developer', 4);
INSERT INTO HumanResources.PossibleSkills (SkillName, Category, Credit)
  VALUES('Project Management', 'Management', 'Two');
GO
```

   The script generates an error on the third INSERT statement. How many new rows do you expect to find in the PossibleSkills table after this error?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Controlling Program Flow

All programming languages include elements that allow you to determine the flow of the program, or the order in which statements are executed. While not as fully featured as languages like C#, T-SQL provides a set of control-of-flow keywords you can use to perform logic tests and create loops containing your T-SQL data manipulation statements. In this lesson, you will learn how to use the T-SQL IF and WHILE keywords.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the control-of-flow elements in T-SQL.
- Write T-SQL code using IF...ELSE blocks.
- Write T-SQL code that uses WHILE.

Understanding T-SQL Control-of-Flow Language

SQL Server provides language elements that control the flow of program execution within T-SQL batches, stored procedures, and multi-statement user-defined functions. These control-of-flow elements allow you to programmatically determine whether or not to execute statements and programmatically determine the order of those statements that should be executed.

- SQL Server provides additional language elements that control the flow of execution of T-SQL statements
  - Used in batches, stored procedures, and multi-statement functions
- Control-of-flow elements allow statements to be performed in a specified order or not at all
  - The default is for statements to execute sequentially
- Includes IF...ELSE, BEGIN...END, WHILE, RETURN, and others

```
IF OBJECT_ID('dbo.t1') IS NOT NULL
        DROP TABLE dbo.t1;
GO
```

These elements include, but are not limited to:

- IF...ELSE, which executes code based on a Boolean expression.
- WHILE, which creates a loop that executes as long as a condition is true.
- BEGIN...END, which defines a series of T-SQL statements that should be executed together.
- Other keywords (for example, BREAK, CONTINUE, WAITFOR, and RETURN), which are used to support T-SQL control-of-flow operations.

You will learn how to use some of these elements in the next lesson.

Additional reading can be found in Books Online at:

Control-of-Flow Language (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402799

Working with IF...ELSE

The IF...ELSE structure is used in T-SQL to conditionally execute a block of code based on a predicate. The IF statement determines whether or not the following statement or block (if BEGIN...END is used) executes. If the predicate evaluates to TRUE, the code in the block is executed. If the predicate evaluates to FALSE or UNKNOWN, the block is not executed, unless the optional ELSE keyword is used to identify another block of code.

- IF...ELSE uses a predicate to determine the flow of the code
  - The code in the IF block is executed if the predicate evaluates to TRUE
  - The code in the ELSE block is executed if the predicate evaluates to FALSE or UNKNOWN
- Very useful when combined with the EXISTS operator

```
IF OBJECT_ID('dbo.t1') IS NULL
        PRINT 'Object does not exist';
ELSE
        DROP TABLE dbo.t1;
GO
```

For example, the following IF statement, without an ELSE, will only execute the statements between BEGIN and END if the predicate evaluates to TRUE, indicating that the object exists. If it evaluates to FALSE or UNKNOWN, no action is taken and execution resumes after the END statement:

IF Example

```
USE TSQL;
GO
IF OBJECT_ID('HR.Employees') IS NULL --this object does exist in the sample database
BEGIN
      PRINT 'The specified object does not exist';
END;
```

With the use of ELSE, you have another execution option when the IF predicate evaluates to FALSE or UNKNOWN, as in the following example:

IF...ELSE Example

```
IF OBJECT_ID('HR.Employees') IS NULL
BEGIN
      PRINT 'The specified object does not exist';
END
ELSE
BEGIN
      PRINT 'The specified object exists';
END;
```

Within data manipulation operations, using IF with the EXISTS keyword can be a useful tool for efficient existence checks, as in the following example:

Existence Check

```
IF EXISTS (SELECT * FROM Sales.EmpOrders WHERE empid =5)
      BEGIN
              PRINT 'Employee has associated orders';
      END;
```

Go to Books Online at:

IF...ELSE (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402800

MCT USE ONLY. STUDENT USE PROHIBITED

Working with WHILE

The WHILE statement is used to execute code in a loop based on a predicate. Like the IF statement, the WHILE statement determines whether the following statement or block (if BEGIN...END is used) executes. The loop ends when the predicate evaluates to FALSE or UNKNOWN. Typically, you control the loop with a variable tested by the predicate and manipulated in the body of the loop itself.

- WHILE enables code to execute in a loop
- Statements in the WHILE block repeat as the predicate evaluates to TRUE
- The loop ends when the predicate evaluates to FALSE or UNKNOWN
- Execution can be altered by BREAK or CONTINUE

```
DECLARE @empid AS INT = 1, @lname AS NVARCHAR(20);
WHILE @empid <=5
  BEGIN
      SELECT @lname = lastname FROM HR.Employees
             WHERE empid = @empid;
      PRINT @lname;
      SET @empid += 1;
  END;
```

The following example uses the @empid variable in the predicate and changes its value in the BEGIN...END block:

WHILE Example

```
DECLARE @empid AS INT = 1, @lname AS NVARCHAR(20);
WHILE @empid <=5
   BEGIN
      SELECT @lname = lastname FROM HR.Employees
             WHERE empid = @empid;
      PRINT @lname;
      SET @empid += 1;
   END;
```

Note: Remember that if SELECT returns UNKNOWN, the variable retains its current value. If there is no employee with an ID equal to @empid, the variable doesn't change from one iteration to another. This would lead to an infinite loop.

The result is:

```
Davis
Funk
Lew
Peled
Buck
```

For additional options within a WHILE loop, you can use the CONTINUE and BREAK keywords to control the flow. For more information about these options, go to Books Online at:

WHILE (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402801

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Controlling Program Flow

In this demonstration, you will see how to:

- Control the flow of execution.

Demonstration Steps

Control the Flow of Execution

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod16\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod16\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

Check Your Knowledge

Question	
You want to populate a table by creating 15 new rows. Before you create the rows, you want to check that the table exists. From the following T-SQL keywords, choose the one that you will NOT need to use.	
Select the correct answer.	
	IF
	WHILE
	BEGIN
	END
	INSERT

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Programming with T-SQL

Scenario

You are a junior database developer for Adventure Works, who has so far focused on writing reports using corporate databases stored in SQL Server 2014. To prepare for upcoming tasks, you will be working with some basic T-SQL programming objects.

Objectives

After completing this lab, you will be able to:

- Declare variables and delimit batches.
- Use control of flow elements.
- Use variables with a dynamic SQL statement.
- Use synonyms

Estimated Time: 45 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Declaring Variables and Delimiting Batches

Scenario

You will practice how to declare variables, retrieve their values, and use them in a SELECT statement to return specific employee information.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Declare a Variable and Retrieve the Value
3. Set the Variable Value Using a SELECT Statement
4. Use a Variable in the WHERE Clause
5. Use Variables with Batches

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab16\Starter folder as Administrator.

▶ Task 2: Declare a Variable and Retrieve the Value

1. In SQL Server Management Studio, open the project file D:\Labfiles\Lab16\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.
2. Write T-SQL code that will create a variable called @num as an int data type. Set the value of the variable to 5 and display it using the alias mynumber. Execute the T-SQL code.
3. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab16\Solution\52 - Lab Exercise 1 - Task 1_1 Result.txt.

4. Write the batch delimiter GO after the written T-SQL code. In addition, write new T-SQL code that defines two variables, @num1 and @num2, both as an int data type. Set the values to 4 and 6 respectively. Write a SELECT statement to retrieve the sum of both variables using the alias totalnum. Execute the T-SQL code.
5. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab16\Solution\53 - Lab Exercise 1 - Task 1_2 Result.txt.

▶ Task 3: Set the Variable Value Using a SELECT Statement

1. Write T-SQL code that defines the variable @empname as an nvarchar(30) data type.
2. Set the value by executing a SELECT statement against the HR.Employees table. Compute a value that concatenates the firstname and lastname column values. Add a space between the two column values and filter the results to return the employee whose empid value is equal to 1.
3. Return the @empname variable's value using the alias employee.
4. Execute the T-SQL code.
5. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab16\Solution\54 - Lab Exercise 1 - Task 2Result.txt.
6. What would happen if the SELECT statement was returning more than one row?

▶ Task 4: Use a Variable in the WHERE Clause

1. Copy the T-SQL code from task 2 and modify it by defining an additional variable named @empid with an int data type. Set the variable's value to 5. In the WHERE clause, modify the SELECT statement to use the newly-created variable as a value for the column empid.
2. Execute the modified T-SQL code.
3. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab16\Solution\55 - Lab Exercise 1 - Task 3 Result.txt.
4. Change the @empid variable's value from 5 to 2 and execute the modified T-SQL code to observe the changes.

▶ Task 5: Use Variables with Batches

1. Copy the T-SQL code from task 4 and modify it by adding the batch delimiter GO before the statement:

```
SELECT @empname AS employee;
```

2. Execute the modified T-SQL code.
3. What happened? What is the error message? Can you explain why the batch delimiter caused an error?

Results: After this exercise, you should know how to declare and use variables in T-SQL code.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Using Control-of-Flow Elements

Scenario

You would like to include conditional logic in your T-SQL code to control the flow of elements by setting different values to a variable using the IF statement.

The main tasks for this exercise are as follows:

1. Write Basic Conditional Logic
2. Check the Employee Birthdate
3. Create and Execute a Stored Procedure
4. Execute a Loop Using the WHILE Statement
5. Remove the Stored Procedure

▶ Task 1: Write Basic Conditional Logic

1. Open the SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. Write T-SQL code that defines the variable @result as an nvarchar(20) data type and the variable @i as an int data type. Set the value of the @i variable to 8. Write an IF statement that implements the following logic:
   - For @i variable values less than 5, set the value of the @result variable to "Less than 5".
   - For @i variable values between 5 and 10, set the value of the @result variable to "Between 5 and 10".
   - For all @i variable values over 10, set the value of the @result variable to "More than 10".
   - For other @i variable values, set the value of the @result variable to "Unknown".
3. At the end of the T-SQL code, write a SELECT statement to retrieve the value of the @result variable using the alias result. Highlight the complete T-SQL code and execute it.
4. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab16\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.
5. Copy the T-SQL code and modify it by replacing the IF statement with a CASE expression to get the same result.

▶ Task 2: Check the Employee Birthdate

1. Write T-SQL code that declares two variables: @birthdate (data type date) and @cmpdate (data type date).
2. Set the value of the @birthdate variable by writing a SELECT statement against the HR.Employees table and retrieving the column birthdate. Filter the results to include only the employee with an empid equal to 5.
3. Set the @cmpdate variable to the value January 1, 1970.
4. Write an IF conditional statement by comparing the @birthdate and @cmpdate variable values. If @birthdate is less than @cmpdate, use the PRINT statement to print the message "The person selected was born before January 1, 1970". Otherwise, print the message "The person selected was born on or after January 1, 1970".
5. Execute the T-SQL code.

6. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab16\Solution\63 - Lab Exercise 2 - Task 2 Result.txt. This is a simple example for the purpose of this exercise. Typically, a different statement block would execute in each case.

▶ Task 3: Create and Execute a Stored Procedure

1. The IT department has provided T-SQL code that encapsulates the previous task in a stored procedure named Sales.CheckPersonBirthDate. It has two parameters: @empid, which you use to specify an employee id, and @cmpdate, which you use as a comparison date. Execute the provided T-SQL code:

```
CREATE PROCEDURE Sales.CheckPersonBirthDate
@empid int,
@cmpdate date
AS
DECLARE
@birthdate date;
SET @birthdate = (SELECT birthdate FROM HR.Employees WHERE empid = @empid);
IF @birthdate < @cmpdate
PRINT 'The person selected was born before ' + FORMAT(@cmpdate, 'MMMM d, yyyy', 'en-
US')
ELSE
PRINT 'The person selected was born on or after ' + FORMAT(@cmpdate, 'MMMM d, yyyy',
'en-US');
```

2. Write an EXECUTE statement to invoke the Sales.CheckPersonBirthDate stored procedure using the parameters of 3 for @empid and January 1, 1990, for @cmpdate. Execute the T-SQL code.
3. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab16\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

▶ Task 4: Execute a Loop Using the WHILE Statement

1. Write T-SQL code to loop 10 times, displaying the current loop information on each occasion.
2. Define the @i variable as an int data type. Write a WHILE statement to execute while the @i variable value is less than or equal to 10. Inside the loop statement, write a PRINT statement to display the value of the @i variable using the alias loopid. Add T-SQL code to increment the @i variable value by 1.
3. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab16\Solution\65 - Lab Exercise 2 - Task 4 Result.txt.

▶ Task 5: Remove the Stored Procedure

1. Execute the provided T-SQL code under the task 5 description to remove the created stored procedure.

Results: After this exercise, you should know how to control the flow of the elements inside the T-SQL code.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Using Variables in a Dynamic SQL Statement

Scenario

You will practice how to invoke dynamic SQL code and how to pass variables to it.

The main tasks for this exercise are as follows:

1. Write a Dynamic SQL Statement That Does Not Use a Parameter

2. Write a Dynamic SQL Statement That Uses a Parameter

▶ Task 1: Write a Dynamic SQL Statement That Does Not Use a Parameter

1. Open the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. Write T-SQL code that defines the variable @SQLstr as nvarchar(200) data type. Set the value of the variable to a SELECT statement that retrieves the empid, firstname, and lastname columns in the HR.Employees table.
3. Write an EXECUTE statement to invoke the written dynamic SQL statement inside the @SQLstr variable. Execute the T-SQL code.
4. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab16\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a Dynamic SQL Statement That Uses a Parameter

1. Copy the previous T-SQL code and modify it to include in the dynamic batch stored in @SQLstr, a filter in which empid is equal to a parameter named @empid. In the calling batch, define a variable named @SQLparam as nvarchar(100). This variable will hold the definition of the @empid parameter. This means setting the value of the @SQLparam variable to @empid int.
2. Write an EXECUTE statement that uses sp_executesql to invoke the code in the @SQLstr variable, passing the parameter definition stored in the @SQLparam variable to sp_executesql. Assign the value 5 to the @empid parameter in the current execution.
3. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab16\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

Results: After this exercise, you should have a basic knowledge of generating and invoking dynamic SQL statements.

Exercise 4: Using Synonyms

Scenario

You will practice how to create a synonym for a table inside the AdventureWorks2008R2 database and how to write a query against it.

The main tasks for this exercise are as follows:

1. Create and Use a Synonym for a Table

2. Drop the Synonym

▶ Task 1: Create and Use a Synonym for a Table

1. Open the T-SQL script 81 - Lab Exercise 4.sql. Ensure that you are connected to the TSQL database.

2. Write T-SQL code to create a synonym named dbo.Person for the Person.Person table in the AdventureWorks database. Execute the written statement.
3. Write a SELECT statement against the dbo.Person synonym and retrieve the FirstName and LastName columns. Execute the SELECT statement.
4. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab16\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.

▶ Task 2: Drop the Synonym

1. Execute the provided T-SQL code under the Task 2 description to remove the synonym.

Results: After this exercise, you should know how to create and use a synonym.

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways

Review Question(s)

Question: Can you declare a variable in one batch and reference it in multiple batches?

Question: Can you create a synonym that references an object that does not exist yet?

Question: Will a WHILE loop exit when the predicate evaluates to NULL?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 17

Implementing Error Handling

Contents:

Module Overview	**17-1**
Lesson 1: Implementing T-SQL Error Handling	**17-2**
Lesson 2: Implementing Structured Exception Handling	**17-7**
Lab: Implementing Error Handling	**17-11**
Module Review and Takeaways	**17-14**

Module Overview

When creating applications for Microsoft® SQL Server® using the T-SQL language, appropriate handling of errors is critically important. A large number of myths surround how error handling in T-SQL works. In this module, you will explore T-SQL error handling; look at how it has traditionally been implemented and how structured exception handling can be used.

Objectives

After completing this module, you will be able to:

- Implement T-SQL error handling.
- Implement structured exception handling.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Implementing T-SQL Error Handling

You should consider how errors can be handled or reported in T-SQL. The T-SQL language offers a variety of error handling capabilities. This lesson covers basic T-SQL error handling, including how you can raise errors intentionally and set up alerts to fire when errors occur. In the next lesson, you will see how to implement a more advanced form of error handling known as structured exception handling.

Lesson Objectives

After completing this lesson, you will be able to:

- Raise errors using the RAISERROR statement.
- Raise errors using the THROW statement.
- Use the @@ERROR system variable.
- Create custom errors.
- Create alerts that fire when errors occur.

Errors and Error Messages

An error indicates a problem or notable issue that arises during a database operation. Each error includes the following elements:

Elements of Database Engine Errors	
Error number	Unique number identifying the specific error
Error message	Text describing the error
Severity	Numeric indication of seriousness from 1-25
State	Internal state code for the database engine condition
Procedure	The name of the stored procedure or trigger in which the error occurred
Line number	Which statement in the batch or procedure generated the error

- System error message are in **sys.messages**
- Add custom application errors using **sp_add_message**

- **Error number** - Unique number identifying the specific error
- **Error message** - Text describing the error
- **Severity** - Numeric indication of seriousness from 1-25
- **State** - Internal state code for the database engine condition
- **Procedure** - The name of the stored procedure or trigger in which the error occurred
- **Line number** - Which statement in the batch or procedure generated the error

Errors can be generated by the SQL Server database engine in response to an event or failure at the system level, or you can generate application errors in your Transact-SQL code.

System Errors

System errors are pre-defined, and you can view them in the **sys.messages** system view. When a system error occurs, SQL Server may take automatic remedial action depending on the severity of the error. For example, when a high-severity error occurs, SQL Server may take a database offline or even stop the database engine service.

Custom Errors

You can generate errors in Transact-SQL code to respond to application-specific conditions or to customize information sent to client applications in response to system errors. These application errors can be defined in-line where they are generated, or you can pre-define them in the **sys.messages** table alongside the system-supplied errors. The error numbers used for custom errors must be 50001 or above.

MCT USE ONLY. STUDENT USE PROHIBITED

To add a custom error message to **sys.messages**, use **sp_addmessage**. The user for the message must be a member of the sysadmin or serveradmin fixed server roles.

This is the sp_addmessage syntax:

sp_addmessage Syntax

```
sp_addmessage [ @msgnum= ] msg_id , [ @severity= ] severity , [ @msgtext= ] 'msg'
     [ , [ @lang= ] 'language' ]
     [ , [ @with_log= ] { 'TRUE' | 'FALSE' } ]
     [ , [ @replace= ] 'replace' ]
```

This is an example of a custom error message:

sp_addmessage Example

```
sp_addmessage 50001, 10, N'Unexpected value entered';
```

As well as being able to define custom error messages, members of the sysadmin server role can also use an additional parameter, @with_log. When set to TRUE, the error will also be recorded in the Windows Application log. Any message written to the Windows Application log is also written to the SQL Server error log. Be judicious with the use of the @with_log option as network and system administrators tend to dislike applications that are "chatty" in the system logs. However, if the error needs to be trapped by an alert, the error must first be written to the Windows Application log.

Note that raising system errors is not supported.

Messages can be replaced without deleting them first by using the @replace = 'replace' option.

The messages are customizable and different ones can be added for the same error number for multiple languages, based on a language_id value. (Note: English messages are language_id 1033).

Raising Errors Using RAISERROR

Both PRINT and RAISERROR can be used to return information or warning messages to applications. RAISERROR allows applications to raise an error that could then be caught by the calling process.

RAISERROR is used to:
- Help troubleshoot T-SQL code
- Check the values of data
- Return messages that contain variable text

```
RAISERROR (N'%s %d', -- Message text.
        10, -- Severity,
        1, -- State,
        N'Custom error message number ',
        2);
```

```
Returns:
Custom error message number 2
```

RAISERROR

The ability to raise errors in T-SQL makes error handling in the application easier, as it is sent like any other system error. RAISERROR is used to:

- Help troubleshoot T-SQL code.
- Check the values of data.
- Return messages that contain variable text.

Note that using a PRINT statement is similar to raising an error of severity 10, as shown in the sample on the slide.

Substitution Placeholders and Message Number

Note that, in the message shown in the example on the slide, %d is a placeholder for a number and %s is a placeholder for a string. Note also that a message number was not mentioned. When errors with message strings are raised using this syntax, they always have error number 50000.

Raising Errors Using THROW

The THROW statement offers a simpler method of raising errors in code. Errors must have an error number of at least 50000.

THROW

THROW differs from RAISERROR in several ways:

- Errors raised by THROW are always severity 16.
- The messages returned by THROW are not related to any entries in sys.sysmessages.
- Errors raised by THROW only cause transaction abort when used in conjunction with SET XACT_ABORT ON and the session is terminated.

Using @@Error

Most traditional error handling code in SQL Server applications has been created using @@ERROR. Note that structured exception handling was introduced in SQL Server 2005 and provides a strong alternative to using @@ERROR. It will be discussed in the next lesson. A large amount of existing SQL Server error handling code is based on @@ERROR, so it is important to understand how to work with it.

@@ERROR

@@ERROR is a system variable that holds the error number of the last error that has occurred. One significant challenge with @@ERROR is that the value it holds is quickly reset as each additional statement is executed.

For example, consider the following code:

@@ERROR Example

```
RAISERROR(N'Message', 16, 1);
IF @@ERROR <> 0
PRINT 'Error=' + CAST(@@ERROR AS VARCHAR(8));
GO
You might expect that, when the code is executed, it would return the error number in a
printed string. However, when the code is executed, it returns:
Msg 50000, Level 16, State 1, Line 1
Message
Error=0
```

Note that the error was raised but the message printed was “Error=0”. In the first line of the output, you can see that the error, as expected, was actually 50000, with a message passed to RAISERROR. This is because the IF statement that follows the RAISERROR statement was executed successfully and caused the @@ERROR value to be reset.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

Capturing @@ERROR Into a Variable

For this reason, when working with @@ERROR, it is important to capture the error number into a variable as soon as it is raised and then continue processing with the variable.

Look at the following code that demonstrates this:

Capturing @@ERROR Into a Variable

```
DECLARE @ErrorValue int;
RAISERROR(N'Message', 16, 1);
SET @ErrorValue = @@ERROR;
IF @ErrorValue <> 0
PRINT 'Error=' + CAST(@ErrorValue AS VARCHAR(8));
```

When this code is executed, it returns the following output:

```
Msg 50000, Level 16, State 1, Line 2
Message
Error=50000
```

Note that the error number is correctly reported now.

Centralizing Error Handling

One other significant issue with using @@ERROR for error handling is that it is difficult to centralize within your T-SQL code. Error handling tends to end up scattered throughout the code. It would be possible to centralize error handling using @@ERROR to some extent, by using labels and GOTO statements. However, this would be frowned upon by most developers today as a poor coding practice.

Creating Alerts When Errors Occur

For certain categories of errors, administrators might create SQL Server Alerts, because they wish to be notified as soon as these occur. This can even apply to user-defined error messages. For example, you may wish to raise an alert whenever a transaction log fills. Alerting is commonly used to bring high severity errors (such as severity 19 or above) to the attention of administrators.

- Alerts can be fired by messages that are stored in the Windows log
- If a message is not normally logged, it can be logged when it is raised with the addition of WITH LOG

Raising Alerts

Alerts can be created for specific error messages. The alerting service works by registering itself as a callback service with the event logging service. This means that alerts only work on logged errors.

There are two ways to make an error raise an alert—you can use the WITH LOG option when raising the error or the message can be altered to make it logged by executing sp_altermessage. The WITH LOG option affects only the current statement. Using sp_altermessage changes the error behavior for all future use. Modifying system errors via sp_altermessage is only possible from SQL Server 2005 SP3 or SQL Server 2008 SP1 onwards.

Demonstration: Handling Errors Using T-SQL

In this demonstration, you will see how to:

- Handle errors.

Demonstration Steps

Handle Errors

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod17\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod17\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

   Question: You are writing some error handling in a T-SQL script. If a problem arises, you want to raise an error with a severity of 20. Should you use RAISERROR or THROW for this error handling?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Implementing Structured Exception Handling

Now you have an understanding of the nature of errors and basic error handling in T-SQL, it is time to look at a more advanced form of error handling. Structured exception handling was introduced in SQL Server 2005. You will see how to use it and evaluate its benefits and limitations.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain TRY CATCH block programming.
- Describe the role of error handling functions.
- Describe catchable versus non-catchable errors.
- Explain how TRY CATCH relates to transactions.
- Explain how errors in managed code are surfaced.

TRY/CATCH Block Programming

Structured exception handling has been part of high level languages for some time, after being introduced to the T-SQL language by SQL Server 2005.

TRY/CATCH Block Programming

Structured exception handling is more powerful than error handling based on the @@ERROR system variable. It allows you to prevent code from being littered with error handling code and to centralize that error handling code.

Centralization of error handling code also allows you to focus more on the purpose of the code rather than the error handling it contains.

TRY Block and CATCH Block

When using structured exception handling, code that might raise an error is placed within a TRY block. TRY blocks are enclosed by BEGIN TRY and END TRY statements.

Should a catchable error occur (most errors can be caught), execution control moves to the CATCH block. The CATCH block is a series of T-SQL statements enclosed by BEGIN CATCH and END CATCH statements.

Note that, while BEGIN CATCH and END TRY are separate statements, the BEGIN CATCH must immediately follow the END TRY.

Current Limitations

High level languages often offer a try/catch/finally construct, and is often used to release resources implicitly. There is no equivalent FINALLY block in T-SQL.

Error Handling Functions

CATCH blocks make the error-related information available throughout the duration of the CATCH block. This includes sub-scopes, such as stored procedures, run from within the CATCH block.

- CATCH blocks make the error-related information available throughout the duration of the CATCH block
- @@ERROR is reset when the next statement is run

Error Handling Functions

You should recall that, when programming with @@ERROR, the value held by the @@ERROR system variable was reset as soon as the next statement was executed.

Another key advantage of structured exception handling in T-SQL is that a series of error handling functions has been provided and these retain their values throughout the CATCH block. Separate functions provide each property of an error that has been raised.

This means you can write generic error handling stored procedures that can still access the error-related information.

Catchable vs. Non-catchable Errors

It is important to realize that, while TRY/CATCH blocks allow you to catch a much wider range of errors than you could with @@ERROR, you cannot catch every type.

- TRY/CATCH blocks will only catch errors in the same block
- Common examples of non-catchable errors are:
  - Compile errors, such as syntax errors, that prevent a batch from compiling
  - Statement level recompilation issues which usually relate to deferred name resolution

Catchable vs. Non-catchable Errors

Not all errors can be caught by TRY/CATCH blocks within the same scope where the TRY/CATCH block exists. Often, errors that cannot be caught in the same scope can be caught in a surrounding scope. For example, you might not be able to catch an error within the stored procedure that contains the TRY/CATCH block. However, you are likely to catch that error in a TRY/CATCH block in the code that called the stored procedure where the error occurred.

Common Non-catchable Errors

Common examples of non-catchable errors are:

- Compile errors, such as syntax errors, that prevent a batch from compiling.
- Statement level recompilation issues which usually relate to deferred name resolution. For example, you could create a stored procedure that refers to an unknown table. An error is only thrown when the procedure tries to resolve the name of the table to an objectid.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Rethrowing Errors Using THROW

If the THROW statement is used in a CATCH block without any parameters, it will rethrow the error that caused the code to enter the CATCH block. You can use this technique to implement error logging in the database by catching errors and logging their details, and then throw the original error to the client application so that it can be handled there.

- Use THROW without parameters to re-raise a caught error
- Must be within a CATCH block

```
BEGIN TRY
        -- code to be executed
END TRY
BEGIN CATCH
        PRINT ERROR_MESSAGE();
        THROW;
END CATCH;
```

In some earlier versions of SQL Server, there was no method to throw a system error. While THROW cannot specify a system error to raise, when THROW is used without parameters in a CATCH block, it will re-raise both system and user errors.

Errors in Managed Code

SQL CLR integration allows for the execution of managed code within SQL Server. High level .NET languages such as C# and VB have detailed exception handling available to them. Errors can be caught using standard .NET try/catch/finally blocks.

- Errors should be handled by managed code
- Unhandled errors will return error number 6522 to the calling T-SQL code

Errors in Managed Code

In general, you may wish to catch errors within managed code as much as possible.

It is important to realize, though, that any errors not handled in the managed code are passed back to the calling T-SQL code. Whenever any error that occurs in managed code is returned to SQL Server, it will appear to be a 6522 error. Errors can be nested and that particular error will be wrapping the real cause of the error.

Another rare but possible cause of errors in managed code would be that the code could execute a RAISERROR T-SQL statement via a SqlCommand object.

Demonstration: Using a TRY/CATCH Block

In this demonstration, you will see how to:

Use a TRY/CATCH block.

Demonstration Steps

Use a TRY/CATCH Block

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod17\Setup.cmd as an administrator.

2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod17\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

Question: You have the following T-SQL script:

```
BEGIN TRY
   INSERT INTO HumanResources.PossibleSkills(SkillName, Category)
   VALUES ('Database Administration', 'IT Professional');
END TRY
DECLARE @prefix AS NVARCHAR(50) = 'There has been an error: ';
BEGIN CATCH
   PRINT @prefix + ERROR_MESSAGE();
   THROW;
END CATCH;
GO
```

The code will not compile and execute. What should you do to troubleshoot this code?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Implementing Error Handling

Scenario

You are a junior database developer for Adventure Works, who will be creating stored procedures using corporate databases stored in SQL Server 2012. In order to create more robust procedures, you will be implementing error handling in your code.

Objectives

After completing this lab, you will be able to:

- Redirect errors with TRY/CATCH.
- Use THROW to pass an error message to a client.

Estimated Time: 30 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Redirecting Errors with TRY/CATCH

Scenario

After this exercise, you should be able to capture and handle errors using a TRY/CATCH construct.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a Basic TRY/CATCH Construct
3. Display an Error Number and an Error Message
4. Add Conditional Logic to a CATCH Block
5. Execute a Stored Procedure in the CATCH Block

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.

   Run Setup.cmd in the D:\Labfiles\Lab17\Starter folder as Administrator.

▶ Task 2: Write a Basic TRY/CATCH Construct

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. Open the project file D:\Labfiles\Lab17\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.
3. Execute the provided SELECT statement:

```
SELECT CAST(N'Some text' AS int);
```

4. Notice that you get an error. Write a TRY/CATCH construct by placing the SELECT statement in a TRY block. In the CATCH block, use the PRINT command to display the text “Error”. Execute the T-SQL code.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Display an Error Number and an Error Message

1. The IT department has provided T-SQL code that looks like this:

```
DECLARE @num varchar(20) = '0';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
END CATCH;
```

2. Execute the provided T-SQL code. Notice that nothing happens although, based on the @num variable's value, you should get an error because of the division by zero. Why didn't you get an error?
3. Modify the CATCH block by adding two PRINT statements. The first statement should display the error number by using the ERROR_NUMBER function. The second statement should display the error message by using the ERROR_MESSAGE function. Also, include a label in each printed message, such as "Error Number:" for the first message and "Error Message:" for the second one.
4. Execute and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab17\Solution\Exercise 1 - Task 2_1 Result.txt.
5. Change the value of the @num variable from 0 to A and execute the T-SQL code.
6. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab17\Solution\Exercise 1 - Task 2_2 Result.txt.
7. Change the value of the @num variable from A to 1000000000 and execute the T-SQL code.
8. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab17\Solution\Exercise 1 - Task 2_3 Result.txt.

▶ Task 4: Add Conditional Logic to a CATCH Block

1. Modify the T-SQL code by including an IF statement in the CATCH block before the added PRINT statements. The IF statement should check to see whether the error number is equal to 245 or 8114. If this condition is true, display the message "Handling conversion error…" using a PRINT statement. If this condition is not true, the message "Handling non-conversion error…" should be displayed.
2. Set the value of the @num variable to A and execute the T-SQL code.
3. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab17\Solution\ Exercise 1 - Task 3_1 Result.txt.
4. Change the value of the @num variable to 0 and execute the T-SQL code.
5. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab17\Solution\Exercise 1 - Task 3_2 Result.txt.

▶ Task 5: Execute a Stored Procedure in the CATCH Block

1. The IT department has provided you with T-SQL code to create a stored procedure named dbo.GetErrorInfo to display different information about the error. Execute the provided T-SQL code:

```
CREATE PROCEDURE dbo.GetErrorInfo AS
PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS varchar(10));
PRINT 'Error Message: ' + ERROR_MESSAGE();
PRINT 'Error Severity: ' + CAST(ERROR_SEVERITY() AS varchar(10));
PRINT 'Error State: ' + CAST(ERROR_STATE() AS varchar(10));
PRINT 'Error Line: ' + CAST(ERROR_LINE() AS varchar(10));
PRINT 'Error Proc: ' + COALESCE(ERROR_PROCEDURE(), 'Not within procedure');
```

MCT USE ONLY. STUDENT USE PROHIBITED

2. Modify the TRY/CATCH code by writing an EXECUTE statement in the CATCH block to invoke the stored procedure dbo.GetErrorInfo.
3. Execute the T-SQL code.

Exercise 2: Using THROW to Pass an Error Message Back to a Client

Scenario

You will practice how to pass an error message using the THROW statement and how to send custom error messages.

The main tasks for this exercise are as follows:

1. Re-Throw the Existing Error Back to a Client
2. Add an Error Handling Routine
3. Add a Different Error Handling Routine
4. Remove the Stored Procedure

▶ Task 1: Re-Throw the Existing Error Back to a Client

1. Open the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL2012 database.
2. Modify the code to include the THROW statement in the CATCH block after the EXECUTE statement. Execute the T-SQL code.

▶ Task 2: Add an Error Handling Routine

1. Modify the T-SQL code by replacing a THROW statement with an IF statement. Write a condition to compare the error number to the value 8134. If this condition is true, the message "Handling division by zero…" should be displayed. Otherwise, display the message "Throwing original error" and add a THROW statement.
2. Execute the T-SQL code.

▶ Task 3: Add a Different Error Handling Routine

1. The IT department has provided you with T-SQL code to create a new variable named @msg and set its value:

```
DECLARE @msg AS varchar(2048);
SET @msg = 'You are doing the module 17 on ' + FORMAT(CURRENT_TIMESTAMP, 'MMMM d,
yyyy', 'en-US') + '. It''s not an error but it means that you are near the final
module!';
```

2. Write a THROW statement and specify the message ID of 50001 for the first argument, the @msg variable for the second argument, and the value 1 for the third argument. Highlight the complete T-SQL code and execute it.
3. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab17\Solution\Exercise 2 - Task 3 Result.txt.

▶ Task 4: Remove the Stored Procedure

1. Execute the provided T-SQL code to remove the stored procedure dbo.GetErrorInfo.

Results: After this exercise, you should know how to throw an error to pass messages back to a client.

Module Review and Takeaways

Review Question(s)

Question: Which error types cannot by caught by structured exception handling?

Question: Can TRY/CATCH blocks be nested?

Question: How can you use THROW outside of a CATCH block?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 18

Implementing Transactions

Contents:

Module Overview	18-1
Lesson 1: Transactions and the Database Engine	18-2
Lesson 2: Controlling Transactions	18-7
Lab: Implementing Transactions	18-12
Module Review and Takeaways	18-16

Module Overview

As you continue to move past SELECT statements and into data modification operations with T-SQL, you must begin to consider how to structure batches containing multiple modification statements and those that might encounter errors. In this module, you will learn how to define transactions to control the behavior of batches of T-SQL statements submitted to Microsoft® SQL Server®. You will also learn how to determine whether a runtime error has occurred after work has begun and whether the work needs to be undone.

Objectives

After completing this module, you will be able to:

- Describe transactions and the differences between batches and transactions.
- Describe batches and how they are handled by SQL Server.
- Create and manage transactions with transaction control language (TCL) statements.
- Use SET XACT_ABORT to define SQL Server's handling of transactions outside TRY/CATCH blocks.

Lesson 1
Transactions and the Database Engine

In this lesson, you will compare simple batches of T-SQL statements to transactions, which allow you to control the behavior of code submitted to SQL Server. You will decide whether special action is needed to respond to a runtime error after work has begun and whether the work needs to be undone.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe a SQL Server database transaction.
- Describe the difference between a batch and a transaction.
- Describe how transactions extend batches.

Defining Transactions

Earlier in this course, you learned that a batch was a collection of T-SQL statements sent to SQL Server as a unit for parsing, optimization, and execution. A transaction extends a batch from a unit submitted to the database engine to a unit of work performed by the database engine. A transaction is a sequence of T-SQL statements performed in an all-or-nothing fashion by SQL Server.

Transactions are commonly created in two ways:

- **Autocommit transactions.** Individual data modification statements (for example, INSERT, UPDATE, and DELETE) submitted separately from other commands are automatically wrapped in a transaction by SQL Server. These single-statement transactions are automatically committed when the statement succeeds or are automatically rolled back when the statement encounters a runtime error.
- **Explicit transactions**. User-initiated transactions are created through the use of TCL commands that begin, commit, or roll back work based on user-issued code. TCL is a subset of T-SQL.
- The primary characteristic of a transaction is that all activity within a transaction's boundaries must either succeed or all fail—no partial completion is permitted. User transactions are typically defined to encapsulate operations that must logically occur together, such as entries into related tables as part of a single business operation.

For example, the following batch inserts data into two tables using two INSERT statements that are part of a single order-processing operation:

```
INSERT INTO dbo.SimpleOrders(custid, empid, orderdate)
   VALUES (68,9,'2006-07-12');
INSERT INTO dbo.SimpleOrderDetails(orderid,productid,unitprice,qty)
   VALUES (1, 2,15.20,20);
GO
```

Business rules might dictate that an order is complete only if the data was successfully inserted into both tables. As you will see in the next lesson, a runtime error in this batch might result in data being inserted

MCT USE ONLY. STUDENT USE PROHIBITED

into one table but not the other. Enclosing both INSERT statements in a user-defined transaction provides the ability to undo the data insertion in one table if the INSERT statement in the other table fails. A simple batch does not provide this capability.

SQL Server manages resources on behalf of transactions while they are active. These resources may include locks and entries in the transaction log to allow SQL Server to undo changes made by the transaction, should a rollback be required.

Additional reading can be found in Books Online at:

Transaction Statements (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402802

The Need for Transactions: Issues with Batches

While batches of T-SQL statements provide a unit of code submitted to the server, they do not include any logic for dealing with partial success when a runtime error occurs, even with the use of structured exception handling's TRY/CATCH blocks.

- Some runtime errors during a batch may result in unacceptable partial success:
  - Part of the batch succeeds and part fails, leaving behind the results from the part of the batch that succeeded
- Simple error handling within a batch cannot repair partial success

```
--Batch without transaction management
BEGIN TRY
        INSERT INTO Sales.Orders ... --Insert succeeds
        INSERT INTO Sales.OrderDetails ... --Insert fails
END TRY
BEGIN CATCH
        --Inserted rows still exist in Sales.Orders Table
        SELECT ERROR_NUMBER()
        ...
END CATCH;
```

The following example illustrates this problem:

Code Without Transaction

```
BEGIN TRY
      INSERT INTO dbo.SimpleOrders(custid,
empid, orderdate)
              VALUES (68,9,'2006-07-12');
      INSERT INTO dbo.SimpleOrders(custid,
empid, orderdate)
              VALUES (88,3,'2006-07-15');
      INSERT INTO dbo.SimpleOrderDetails(orderid,productid,unitprice,qty)
              VALUES (1, 2,15.20,20);
      INSERT INTO dbo.SimpleOrderDetails(orderid,productid,unitprice,qty)
              VALUES (999,77,26.20,15);
END TRY
BEGIN CATCH
      SELECT ERROR_NUMBER() AS ErrNum, ERROR_MESSAGE() AS ErrMsg;
END CATCH;
```

If the first INSERT statement succeeds but a subsequent one fails, the new row in the dbo.SimpleOrders table will persist after the end of the batch, even after the execution branches to the CATCH block. This issue applies to any successful statements, if a later statement fails with a runtime error.

Note: Remember that syntax or name-resolution errors cause the entire batch to return an error, preventing any execution. Runtime errors only occur once the batch has been submitted, parsed, planned, and compiled for execution.

To work around this situation, you will need to direct SQL Server to treat the batch as a transaction. You will learn more about creating transactions in the next topic.

MCT USE ONLY STUDENT USE PROHIBITED

Transactions Extend Batches

As you have seen, runtime errors encountered during the execution of simple batches create the possibility of partial success, which is not typically a desired outcome. To address this, you will add code to identify the batch as a transaction by placing the batch between BEGIN TRANSACTION and COMMIT TRANSACTION statements. You will also add error-handling code to roll back the transaction should an error occur. This error-handling code will undo the partial changes made before the error occurred.

- Transaction commands identify blocks of code that must succeed or fail together and provide points where the database engine can roll back, or undo, operations:

```
BEGIN TRY
        BEGIN TRANSACTION
                INSERT INTO Sales.Orders ... --Insert succeeds
                INSERT INTO Sales.OrderDetails ... --Insert fails
        COMMIT TRANSACTION -- If no errors, transaction
            -- completes
END TRY
BEGIN CATCH
    --Inserted rows still exist in Sales.Orders Table
    SELECT ERROR_NUMBER()
    ROLLBACK TRANSACTION --Any transaction work undone
END CATCH;
```

The following example shows the addition of TCL commands to address the possibility of an error occurring after some work has been performed:

Transaction Example

```
BEGIN TRY
 BEGIN TRANSACTION;
      INSERT INTO dbo.SimpleOrders(custid, empid, orderdate)
             VALUES (68,9,'2006-07-15');
      INSERT INTO dbo.SimpleOrderDetails(orderid,productid,unitprice,qty)
             VALUES (99, 2,15.20,20);
      COMMIT TRANSACTION;
END TRY
BEGIN CATCH
 SELECT ERROR_NUMBER() AS ErrNum, ERROR_MESSAGE() AS ErrMsg;
 ROLLBACK TRANSACTION;
END CATCH;
```

Within the TRY block, the INSERT statements are wrapped by BEGIN TRANSACTION and COMMIT TRANSACTION statements. This identifies the INSERT statements as a single unit of work that must succeed or fail together. If no runtime error occurs, the transaction commits, and the result of each INSERT is allowed to persist in the database.

If an error occurs during the execution of the first INSERT statement, the execution branches to the CATCH block, bypassing the second INSERT statement. The ROLLBACK statement in the CATCH block terminates the transaction, releasing its resources.

If an error occurs during the execution of the second INSERT statement, the execution branches to the CATCH block. Because the first INSERT completed successfully and added rows to the dbo.SimpleOrders table, the ROLLBACK statement is used to undo the successful INSERT operation.

Note: You will learn how to use the BEGIN TRANSACTION, COMMIT TRANSACTION, and ROLLBACK TRANSACTION statements in the next lesson.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Transactions and the Database Engine

In this demonstration, you will see how to:

- Use transactions.

Demonstration Steps

Use Transactions

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod18\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod18\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

Categorize Activity

Categorize each T-SQL keyword into the appropriate category. Indicate your answer by writing the category number to the right of each item.

Items	
1	BEGIN TRY
2	COMMIT TRANSACTION
3	INSERT
4	BEGIN TRANSACTION
5	END CATCH
6	RAISERROR
7	END TRANSACTION
8	ROLLBACK TRANSACTION
9	BEGIN CATCH
10	END TRY

Category 1		Category 2
TCL		Non-TCL

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Controlling Transactions

To control how SQL Server treats your data modification statements, you need to use TCL statements. By enclosing batches between BEGIN TRANSACTION and COMMIT or ROLLBACK TRANSACTION statements, you will identify the units of work to be performed together and provide points of recovery in your code.

Lesson Objectives

After completing this lesson, you will be able to:

- Mark the beginning of units of work with BEGIN TRANSACTION.
- Mark successful completion of batches with COMMIT TRANSACTION.
- Undo failed transactions with ROLLBACK TRANSACTION.
- Describe how to use XACT_ABORT to automatically roll back failed T-SQL statements.

BEGIN TRANSACTION

SQL Server will automatically wrap individual data modification statements (for example, INSERT, UPDATE, and DELETE) in their own transactions, which auto-commit on success and auto-rollback on failure. While this behavior is transparent to the user, you have seen the results of this when you have executed a batch of T-SQL statements with partial success. Successful INSERTS have written their values to the target tables, while failed statements have not left values behind.

- BEGIN TRANSACTION marks the starting point of an explicit, user-defined transaction
- Transactions last until a COMMIT statement is issued, a ROLLBACK is manually issued, or the connection is broken and the system issues a ROLLBACK
- Transactions are local to a connection and cannot span connections
- In your T-SQL code: Mark the start of the transaction's work

```
BEGIN TRY
        BEGIN TRANSACTION -- marks beginning of work
                INSERT INTO Sales.Orders ... --transacted work
                INSERT INTO Sales.OrderDetails ... --transacted work
...
```

If you need to identify a group of statements as a transactional unit of work, you cannot rely on this automatic behavior. Instead, you will need to manually specify the boundaries of the unit. To mark the start of a transaction, use the BEGIN TRANSACTION statement, which may also be stated as BEGIN TRAN.

If you are using T-SQL structured exception handling, you will want to begin the transaction inside a TRY block. Within the exception handler, you may decide whether to COMMIT or ROLLBACK the transaction, depending on its outcome.

When you identify your own transactions with BEGIN TRANSACTION, consider the following:

- Once you initiate a transaction, you must properly end it. Use COMMIT TRANSACTION on success or ROLLBACK TRANSACTION on failure.
- While transactions may be nested, inner transactions will be rolled back, even if committed, if the outer transaction rolls back. Therefore, nested transactions are not typically useful in user code.
- Transactions last until a COMMIT TRANSACTION or a ROLLBACK TRANSACTION is issued, or until the originating connection is dropped, at which point SQL Server will roll back the transaction automatically.
- A transaction's scope is the connection in which it was started. Transactions cannot span connections (except by bound sessions, a deprecated feature that is beyond the scope of this course).

- SQL Server may take and hold locks on resources during the lifespan of the transaction. To reduce concurrency issues, consider keeping your transactions as short as possible. See Microsoft course 20464C: *Developing Microsoft SQL Server 2014 Databases* for more information on locking in SQL Server.

Additional reading on BEGIN TRANSACTION statements can be found in Books Online at:

BEGIN TRANSACTION (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402856

Guidance on the use of nested transactions can be found in Books Online at:

Nesting Transactions

http://go.microsoft.com/fwlink/?LinkID=402857

COMMIT TRANSACTION

Once the statements in your transaction have completed without error, you need to instruct SQL Server to end the transaction, making the modifications permanent and releasing resources that were held on behalf of the transaction. To do this, use the COMMIT TRANSACTION (or COMMIT TRAN) statement.

If you are using T-SQL structured exception handling, you will want to COMMIT the transaction inside the TRY block in which you began it.

The following example shows the use of COMMIT TRANSACTION to mark a batch as completed:

COMMIT TRANSACTION Example

```
BEGIN TRY
 BEGIN TRANSACTION
      INSERT INTO dbo.SimpleOrders(custid, empid, orderdate)
      VALUES (68,9,'2006-07-12');
      INSERT INTO dbo.SimpleOrderDetails(orderid,productid,unitprice,qty)
      VALUES (1, 2,15.20,20);
 COMMIT TRANSACTION
END TRY
```

Note: The previous example does not contain logic to determine if the transaction should be committed or rolled back. It is relying on the success of the statements to provide the logic to implement error handling.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

ROLLBACK TRANSACTION

To end a failed transaction, you will use the ROLLBACK command. ROLLBACK undoes any modifications made to data during the transaction, reverting it to the state it was in when the transaction started. This includes rows inserted, deleted, or updated, as well as objects created. ROLLBACK also allows SQL Server to release resources, such as locks, held during the transaction's lifespan.

- A ROLLBACK statement undoes all modifications made in the transaction by reverting the data to the state it was in at the beginning of the transaction
- ROLLBACK frees resources, such as locks, held by the transaction
- Before rolling back, you can test the state of the transaction with the XACT_STATE function
- In your T-SQL code: If an error occurs, ROLLBACK to the point of the BEGIN TRANSACTION statement

```
BEGIN CATCH
        SELECT ERROR_NUMBER() --sample error handling
        ROLLBACK TRAN
END CATCH
```

If you are using T-SQL structured exception handling, you will want to ROLLBACK the transaction inside the CATCH block that follows the TRY block containing the BEGIN and COMMIT statements.

The following example shows the use of the ROLLBACK TRANSACTION statement inside a CATCH block, where the transaction will only be rolled back in case of an error:

ROLLBACK TRANSACTION Example

```
BEGIN TRY
 BEGIN TRANSACTION;
      INSERT INTO dbo.SimpleOrders(custid, empid, orderdate)
      VALUES (68,9,'2006-07-12');
      INSERT INTO dbo.SimpleOrderDetails(orderid,productid,unitprice,qty)
      VALUES (1, 2,15.20,20);
 COMMIT TRANSACTION;
END TRY
BEGIN CATCH
      SELECT ERROR_NUMBER() AS ErrNum, ERROR_MESSAGE() AS ErrMsg;
      ROLLBACK TRANSACTION;
END CATCH;
```

Before issuing a ROLLBACK command, you may wish to test to see if a transaction is active. You can use the T-SQL XACT_STATE function to determine if there is an active transaction to be rolled back. This can help avoid errors being raised inside the CATCH block.

XACT_STATE returns the following values:

XACT_STATE Results	Description
0	There is no active user transaction.
1	The current request has an active, committable, user transaction.
-1	The current request has an active user transaction, but an error has occurred. The transaction can only be rolled back.

The following example shows the use of XACT_STATE to issue a ROLLBACK statement only if the transaction is active but cannot be committed:

```
BEGIN TRY
 BEGIN TRANSACTION;
   INSERT INTO dbo.SimpleOrders(custid, empid, orderdate)
   VALUES (68,9,'2006-07-12');
   INSERT INTO dbo.SimpleOrderDetails(orderid,productid,unitprice,qty)
   VALUES (1, 2,15.20,20);
 COMMIT TRANSACTION;
END TRY
BEGIN CATCH
   SELECT ERROR_NUMBER() AS ErrNum, ERROR_MESSAGE() AS ErrMsg;
   IF (XACT_STATE()) = -1
          BEGIN
        ROLLBACK TRANSACTION;
          END;
   ELSE .... -- provide for other outcomes of XACT_STATE()
END CATCH;
```

See *Transaction Statements (Transact-SQL)* in Books Online at:

Transaction Statements (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402802

Using XACT_ABORT

As you have seen, SQL Server does not automatically roll back transactions when errors occur. In this module, most of the discussion about controlling transactions has assumed the use of TRY/CATCH blocks to perform the logic and either commit or roll back a transaction. For situations in which you are not using TRY/CATCH blocks, another option exists for automatically rolling back a transaction when an error occurs. The XACT_ABORT setting can be used to specify whether SQL Server rolls back the current transaction when a runtime error occurs during the execution of T-SQL code.

- SQL Server does not automatically roll back transactions when errors occur
- To roll back, either use ROLLBACK statements in error-handling logic or enable XACT_ABORT
- XACT_ABORT specifies whether SQL Server automatically rolls back the current transaction when a runtime error occurs
  - When SET XACT_ABORT is ON, the entire transaction is terminated and rolled back on error, unless occurring in TRY block
  - SET XACT_ABORT OFF is the default setting
- Change XACT_ABORT value with the SET command:

```
SET XACT_ABORT ON;
```

By default, XACT_ABORT is off. Change the XACT_ABORT setting with the SET command:

```
SET XACT_ABORT ON;
```

When SET XACT_ABORT is ON, the entire transaction is terminated and rolled back on error, unless the error occurs in a TRY block. An error in a TRY block leaves the transaction open but not committable, despite the setting of XACT_ABORT.

See Books Online at:

SET XACT_ABORT (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402858

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Controlling Transactions

In this demonstration, you will see how to:

- Control transactions.

Demonstration Steps

Control Transactions

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod18\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod18\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

   Question: You have executed the following batch of T-SQL statements:

```
BEGIN TRY
   BEGIN TRANSACTION;
      INSERT INTO dbo.SimpleOrders(custid, empid, orderdate)
         VALUES (68,9,'2006-07-12');
      INSERT INTO
dbo.SimpleOrderDetails(orderid,productid,unitprice,qty)
         VALUES (1, 2,15.20,20);
END TRY
BEGIN CATCH
   SELECT ERROR_NUMBER() AS ErrNum, ERROR_MESSAGE() AS ErrMsg;
   ROLLBACK TRANSACTION;
END CATCH;
```

   A fellow query writing is now receiving errors resulting from locks on database records. What can you do to troubleshoot this problem?

Lab: Implementing Transactions

Scenario

You are a junior database developer for Adventure Works, who will be creating stored procedures using corporate databases stored in SQL Server 2014. To create more robust procedures, you will be implementing transactions in your code.

Objectives

After completing this lab, you will be able to:

- Control transactions.
- Add error handling to a CATCH block.

Estimated Time: 30 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Controlling Transactions with BEGIN, COMMIT, and ROLLBACK

Scenario

The IT department has supplied different examples of INSERT statements to practice executing multiple statements inside one transaction. You will practice how to start a transaction, commit or abort it, and return the database to its state before the transaction.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Commit a Transaction
3. Delete the Previously Inserted Rows from the HR.Employees Table
4. Open a Transaction and Use the ROLLBACK Statement
5. Clear the Modifications Against the HR.Employees Table

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab18\Starter folder as Administrator.

▶ Task 2: Commit a Transaction

1. In SQL Server Management Studio, open the project file D:\Labfiles\Lab18\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

2. The IT department has provided the following T-SQL code:

```
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'20110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
```

3. This code inserts two rows into the HR.Employees table. By default, SQL Server treats each individual statement as a transaction. In other words, by default, SQL Server automatically commits the transaction at the end of each individual statement. So in this case the default behavior would be two transactions since you have two INSERT statements. (Do not worry about the details of the INSERT statements because they are only meant to provide sample code for the transaction scenario).
4. In this example, you would like to control the transaction and execute both INSERT statements inside one transaction.
5. Before the supplied T-SQL code, write a statement to open a transaction. After the supplied INSERT statements, write a statement to commit the transaction. Highlight all of the T-SQL code and execute it.
6. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab18\Solution\52 - Lab Exercise 1 - Task 1_1 Result.txt.
7. Write a SELECT statement to retrieve the empid, lastname, and firstname columns from the HR.Employees table. Order the employees by the empid column in a descending order. Execute the SELECT statement.
8. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab18\Solution\53 - Lab Exercise 1 - Task 1_2 Result.txt. Notice the two new rows in the result set.

▶ Task 3: Delete the Previously Inserted Rows from the HR.Employees Table

1. Execute the provided T-SQL code to delete rows inserted from the previous task:

```
DELETE HR.Employees
WHERE empid IN (10, 11);
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

2. Note that this is cleanup code that will not be explained in this course.

▶ Task 4: Open a Transaction and Use the ROLLBACK Statement

1. The IT department has provided T-SQL code (which happens to the same code as in task 1). Before the provided T-SQL code, write a statement to start a transaction.
2. Highlight the written statement and the provided T-SQL code, and execute it.
3. Write a SELECT statement to retrieve the empid, lastname, and firstname columns from the HR.Employees table. Order the employees by the empid column.
4. Execute the written SELECT statement and notice the two new rows in the result set.
5. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab18\Solution\54 - Lab Exercise 1 - Task 3_1 Result.txt.

6. After the written SELECT statement, write a ROLLBACK statement to cancel the transaction. Only execute the ROLLBACK statement.
7. Highlight and again execute the written SELECT statement against the HR.Employees table.
8. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab18\Solution\55 - Lab Exercise 1 - Task 3_2 Result.txt. Notice that the two new rows are no longer present in the table.

▶ Task 5: Clear the Modifications Against the HR.Employees Table

1. Execute the provided T-SQL code:

```
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

Results: After this exercise, you should be able to control a transaction using the BEGIN TRAN, COMMIT, and ROLLBACK statements.

Exercise 2: Adding Error Handling to a CATCH Block

Scenario

In the previous module, you learned how to add error handling to T-SQL code. Now you will practice how to properly control a transaction by testing to see if an error occurred.

The main tasks for this exercise are as follows:

1. Observe the Provided T-SQL Code
2. Delete the Previously Inserted Row in the HR.Employees Table
3. Abort Both INSERT Statements If an Error Occurs
4. Clear the Modifications Against the HR.Employees Table

▶ Task 1: Observe the Provided T-SQL Code

1. Open the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. The IT department has provided T-SQL code that is similar to the code in the previous exercise:

```
SELECT empid, lastname, firstname
FROM HR.Employees
ORDER BY empid DESC;
GO
BEGIN TRAN;
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'10110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
COMMIT TRAN;
```

3. Execute only the SELECT statement.
4. Observe and compare the results that you achieved with the desired results shown in the file 62 - Lab Exercise 2 - Task 1_1 result.txt. Notice the number of employees in the HR.Employees table.

MCT USE ONLY STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

5. Execute the part of the T-SQL code that starts with a BEGIN TRAN statement and ends with the COMMIT TRAN statement. You will get a conversion error in the second INSERT statement.
6. Again, execute only the SELECT statement.
7. Observe and compare the results that you achieved with the desired results shown in the file 63 - Lab Exercise 2 - Task 1_2 Result.txt. Notice that, although an error showed inside the transaction block, one new row was added to the HR.Employees table based on the first INSERT statement.

▶ Task 2: Delete the Previously Inserted Row in the HR.Employees Table

1. Execute the provided T-SQL code to delete the row inserted from the previous task:

```
DELETE HR.Employees
WHERE empid IN (10, 11);DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

▶ Task 3: Abort Both INSERT Statements If an Error Occurs

1. Modify the provided T-SQL code to include a TRY/CATCH block that rolls back the entire transaction if any of the INSERT statements throws an error:

```
BEGIN TRAN;
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'10110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
COMMIT TRAN;
```

2. In the CATCH block, include a PRINT statement that prints the message "Rollback the transaction…" if an error occurred and the message "Commit the transaction…" if no error occurred.
3. Execute the modified T-SQL code.
4. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab18\Solution\64 - Lab Exercise 2 - Task 3_1 Result.txt.
5. Write a SELECT statement against the HR.Employees table to see if any new rows were inserted (like you did in exercise 1). Execute the SELECT statement.
6. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab18\Solution\65 - Lab Exercise 2 - Task 3_2 Result.txt.

▶ Task 4: Clear the Modifications Against the HR.Employees Table

1. Execute the provided T-SQL code:

```
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

Results: After this exercise, you should have a basic understanding of how to control a transaction inside a TRY/CATCH block to efficiently handle possible errors.

Module Review and Takeaways

Review Question(s)

Question: What happens to a nested transaction when the outer transaction is rolled back?

Question: When a runtime error occurs in a transaction and SET XACT_ABORT is ON, is the transaction always automatically rolled back?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 19

Appendix A: Improving Query Performance

Contents:

Module Overview	**19-1**
Lesson 1: Factors in Query Performance	**19-2**
Lesson 2: Displaying Query Performance Data	**19-10**
Lab: Improving Query Performance	**19-16**
Module Review and Takeaways	**19-20**

Module Overview

In this course, you have learned many techniques for retrieving data using T-SQL queries. You also need to be aware of the performance impact of your queries. This module presents several key guidelines for writing well-performing queries, as well as ways to monitor the execution of your queries and their impact on Microsoft® SQL Server®.

Objectives

After completing this module, you will be able to:

- Describe components of well-performing queries.
- Describe the role of indexes and statistics in SQL Server.
- Display and interpret basic query plans.
- Display and interpret basic query performance data.

Lesson 1
Factors in Query Performance

SQL Server performance offers an extremely broad set of topics that commonly involve the use of tools and concepts beyond the scope of this course. In this lesson, you will learn about those aspects of SQL Server performance that pertain to writing well-performing queries. You will also learn about the basics of indexes, distribution statistics, and how to avoid cursors in your queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe components of well-performing queries.
- Describe how SQL Server uses clustered and nonclustered indexes.
- Describe how SQL Server uses distribution statistics.
- Compare the use of cursors to sets in queries.

Writing Well-Performing Queries

While SQL Server performance has many variables, including central processing units (CPUs), memory, storage subsystems, and networks, you can have an impact on server performance even at the level of a single query. Here are some considerations for writing well-performing queries:

- Only retrieve what you need. Asking for more data than you require increases the I/O footprint of your query without adding benefit. Specifically, avoid the use of the * alias to request all columns, and filter rows by providing a WHERE clause.

Note: In addition to avoiding the use of * for performance reasons, it is recommended that you avoid using * to protect your application code from changes to the design of the underlying tables or views.

- Improve the performance of WHERE clauses by avoiding search operators that don't perform well (such as <>) and by avoiding manipulation of column values in the predicate.

For example, the following query can be rewritten to avoid manipulating a column in an expression for better search performance:

Original Query

```
SELECT empid, hiredate
FROM hr.employees
WHERE DATEDIFF(yy, hiredate, GETDATE())>=8;
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The following example shows the query rewritten for better search performance. The column has been separated from the function:

Query With Improved Performance

```
SELECT empid, hiredate
FROM hr.employees
WHERE hiredate <= DATEADD(yy, -8, GETDATE());
```

- Minimize the use of temporary tables or table variables in your queries. In this course, you have learned how to use window functions and table expressions as an alternative to queries that create and populate temporary tables before further manipulating them.
- Avoid cursors and other iterative approaches that resemble procedural languages, in favor of set-based operations. You will learn more about cursors and how to avoid them later in this module.
- If you are a report writer, consider monitoring the performance of your queries (with tools you will learn about in this module) and working with your database administrators to arrange good indexes that will support your queries. If you will also be acting in a database development or administration role, you will need to learn more about indexing in SQL Server than this course will provide. See Microsoft course 20464D: *Developing Microsoft SQL Server Databases* for more information.
- Learn how to address tasks with different query approaches. This will enable you to compare the performance of each approach and have alternative queries that might perform better than others in differing conditions, such as when the server load varies.

See *Query Performance* in Books Online at:

Query Performance

http://go.microsoft.com/fwlink/?LinkID=402860

Indexing in SQL Server

SQL Server uses indexes to improve the performance of queries when they are searching and filtering rows, and when they are ordering data (for example, grouping, joining, or sorting). When examining query performance, you will find it useful to have a basic understanding of how SQL Server uses indexes to interpret the results of query execution plans. Later in this module, you will learn how to view and interpret query execution plans, which may contain references to index use. This topic is designed to introduce indexes and provide enough information to enable you to interpret basic execution plans.

- SQL Server accesses data by using indexes or by scanning all rows in a table
- Indexes also supports ordering operations such as grouping, joining, and ORDER BY clauses

Note: This lesson is intended only to provide an introduction to indexing concepts. For more information on index design, creation, and maintenance, see Microsoft course 20464D: Developing Microsoft SQL Server Databases.

When executing a query, SQL Server will either read all rows in the source table(s) or use a relevant index to locate the required data. When no index is defined or none is useful to the query optimizer (due to columns indexed, distribution statistics, or other factors), SQL Server must examine each row in the table. This is called a table scan.

For example, the following query is not selective, so SQL Server will scan the entire table to return results:

Non-Selective Query

```
SELECT  empid, lastname, firstname, title, hiredate
FROM hr.employees;
```

When an index is defined and considered useful by the query optimizer, SQL Server can use the index to locate the data requested by the query. If an individual row location is retrieved by the index, the operation is called an index seek; if a range of values is retrieved, this is an index scan.

The following query is highly selective. If an index is defined and useful to SQL Server, such as index on the empid column, SQL Server might use it to perform an index seek to return the results.

Selective Query

```
SELECT  empid, lastname, firstname, title, hiredate
FROM hr.employees
WHERE empid = 7;
```

Note: There are additional considerations beyond the presence or absence of an index to determine whether SQL Server will use an index to solve a query. The purpose of these examples is to illustrate the terminology, not to cover all possible scenarios.

SQL Server Index Basics: Clustered Indexes

In SQL Server, a table may have no indexes, or it may have one or more. Indexes are not required for data access, although there are some features, such as constraints, that create indexes to support them and cannot be removed without dropping the constraint.

If a table has no clustered indexes, it is said to be a heap. In a heap, there is no order to the way the table's rows are organized.

- Clustered indexes determine the logical order of rows within a table
  - Conceptually, a table with a clustered index is like a dictionary, whose terms are the index key
- Characteristics of Clustered Indexes:
  - A clustered index on a column causes a table to be stored with rows logically organized by that column's values
  - A clustered index is not a separate physical structure from the table—index data is stored with the table
  - One clustered index per table

Note: For more information on table structures and objects such as data and index pages, see Microsoft course 20464D: *Developing Microsoft SQL Server Databases*.

A database developer or administrator may choose to add a clustered index to a table. A clustered index causes the rows in the table to be logically stored in order of the column(s) specified in the index, called the index key. The columns used as the index key in a clustered index are called the clustering key. Tables with clustered indexes are maintained in index order, and rows are inserted into the correct logical location determined by their index key value. Rows are stored with their index key value. Clustered indexes are not stored as separate structures in the database.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Since the clustered index determines the order of the rows in the table, only one clustered index is permitted per table.

When SQL Server searches for and locates an index key value in the clustered index, it also locates data for that row. No additional navigation is required, except in the case of special data types. Conceptually, a table with a clustered index is like a dictionary, whose terms are the index key. The terms appear in the dictionary in alphabetical order. When you search the dictionary for a term and locate it, you are also at the definition for the term.

Since there is only one clustered index permitted per table, care should be taken when choosing the column(s) used as the clustering key. See *Create Clustered Indexes* in Books Online at:

Create Clustered Indexes

http://go.microsoft.com/fwlink/?LinkID=402861

SQL Server Index Basics: Nonclustered Indexes

In addition to clustered indexes, SQL Server supports another common type, called a nonclustered index. Nonclustered indexes are separate structures that contain one or more columns as a key, as well as a pointer back to the source row in the table. Nonclustered indexes may be created on tables that are heaps, or on tables organized by clustered indexes. The pointer information stored in the nonclustered index depends on whether the underlying table is organized as a heap or by a clustered index, but is otherwise transparent to a query.

- Nonclustered indexes are separate structures with pointers back to the location of the data
  - Conceptually similar to a subject index printed at the back of a book
- Nonclustered indexes provide alternate ways to rapidly locate data
  - If table's clustered index is on empid, a nonclustered index on last name may be useful for queries that use lastname in the predicate
- A table may have multiple nonclustered indexes
  - Adding nonclustered indexes adds to storage requirements for database, adds to processing time when data is updated

Since the nonclustered index is a separate structure and stores only key data and a pointer, queries that use a nonclustered index may require an additional lookup operation before accessing the underlying data. Conceptually, a nonclustered index is like a subject index printed at the back of a book. A sorted list of items appears, along with a page number that points to the location of the subject in the main section of the book. When a key value is located in a nonclustered index, SQL Server may use the pointer to locate the data in the table itself.

Nonclustered indexes are often added to tables to improve specific query performance and avoid resource-intensive table scans. However, nonclustered indexes require additional disk space for storage, and are updated in real time as the underlying data changes, adding to the duration of transactions. For these reasons, database administrators may decide not to add nonclustered indexes on every column referenced by a query.

See *Create Nonclustered Indexes* in Books Online at:

Create Nonclustered Indexes

http://go.microsoft.com/fwlink/?LinkID=402862

SQL Server Indexes: Performance Considerations

- Check query execution plans to see if indexes are present and being used as expected
- For query writers who are not DBAs or database developers, being able to spot problems with indexes, such as table scans when you expect an index to be used, can be very helpful in tuning an application

For report and query writers, having a basic understanding of indexes will be useful in identifying poorly performing queries. In the next lesson, you will learn how to display and interpret query execution plans, which may include symbols indicating index usage. Being able to identify problems with indexes, such as spotting a table scan when you expected an index to be used, can be very helpful in tuning an application. Similarly, SQL Server might opt to scan an entire table when you expect the use of a nonclustered index. Understanding that nonclustered indexes involve an additional lookup operation versus scanning the table in a single pass, might help you understand the choices of the optimizer.

See the section *How Indexes are Used by the Query Optimizer* in the topic *Clustered and Nonclustered Indexes Described* in Books Online at:

Clustered and Nonclustered Indexes Described

http://go.microsoft.com/fwlink/?LinkID=402864

Distribution Statistics

- Distribution statistics describe the distribution and the uniqueness, or selectivity, of data
- Statistics, by default, are created and updated automatically
- Statistics are used by the query optimizer to estimate the selectivity of data, including the size of the results
- Large variances between estimated and actual values might indicate a problem with the estimates, which may be addressed through updating statistics

SQL Server creates and maintains statistics on the distribution of values in columns, in tables. Distribution statistics are used by the query optimizer to estimate the number of rows involved in a query (selectivity). The optimizer uses this information to help determine the optimal access method for the query. For example, statistics about the number of rows in a table might cause SQL Server to seek through an index rather than scan the entire table.

By default, distribution statistics are automatically created and maintained by SQL Server for:

- The leading key column in all indexes.
- Any column used as a predicate as part of a WHERE clause or JOIN ON clause.
- Columns used in temp tables (although not in table variables).

Administrators and database developers may also manually create statistics and update them manually or by way of scheduled jobs.

Note: Manually creating, updating, and reviewing statistics is beyond the scope of this course. For more information, see Microsoft course 20464D: *Developing Microsoft SQL Server Databases*.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

As a query writer, you will be able to see the effect of statistics by viewing estimates in execution plans. Later in this module, you will learn how to view estimated, as well as actual, execution plans. Estimated execution plans, as the name might imply, are based on estimates of row counts, derived from the distribution statistics. Large variances between estimated and actual values might indicate a problem with the estimates, which may be addressed through updating statistics. Being able to bring this to the attention of the database administrator is a useful skill.

See the technical article *Statistics Used by the Query Optimizer in Microsoft SQL Server 2008* on MSDN at:

Statistics Used by the Query Optimizer in Microsoft SQL Server 2008

http://go.microsoft.com/fwlink/?LinkID=402865

Defining Cursors

So far in this course, you have learned a set-based approach for working with data in SQL Server. However, many people who are new to T-SQL come from a procedural development background and seek a familiar methodology for working with one row at a time. In this topic, you will see how cursors can fill that need. However, you should continue to work on developing set-based approaches for most situations.

- Cursors provide a mechanism for working with one row at a time in a certain order
  - Unlike sets, which allow working on all rows at once and have no order
- Conceptually similar to a WHILE loop cycling through data one row at a time.
- The cursor approach is often a familiar one to new T-SQL developers, but has its drawbacks

SQL Server provides server-side cursors, as opposed to client API cursors such as those provided by OLEDB, for example. Server-side cursors allow applications to work with a single row at a time in a certain order, as against the all-at-once, unordered, set-based approach.

A cursor operates on the result of a SELECT statement and moves through the results, one row at a time. Conceptually, a cursor behaves in a similar fashion to a WHILE loop, cycling through data one step at a time.

The general steps for working with a cursor in T-SQL are as follows:

1. Declare variables to hold the data manipulated by the cursor.
2. Declare a variable of type cursor and assign to it the results of a SELECT statement.
3. Open the cursor.
4. Move to the next row, fetch its data, and operate on it.
5. Repeat step 4 until the end of the rows is reached.
6. Close and de-allocate the cursor.

While this approach may present a familiar, granular method for working in SQL Server, it is best reserved for scenarios in which a set-based approach is not feasible.

See Books Online at:

Cursors

http://go.microsoft.com/fwlink/?LinkID=402866

Avoiding Cursors

While cursors do provide a familiar row-at-a-time-in-order methodology to some, generally they should be avoided for most SQL Server query tasks. Here are some reasons why you should avoid cursors:

- Iterative approaches such as cursors contradict the relational model, which operates on unordered sets. With sets, you tell SQL Server what you want to work with. With cursors you specify how you want the engine to do the work.
- Cursors typically require more code than a set-based approach. Not only is a SELECT statement required to supply data to the cursor, but the cursor requires loop management, navigation, and state-checking code.
- Cursors typically incur more overhead than a set operation with comparable output. While the set operation acts on all rows in the set at once, the cursor must repeat the same row-by-row manipulation for each row accessed by the cursor. Cursors can be an order of magnitude slower (or more) than a set-based operation.

Many of the techniques and query components you have learned to use in this course, such as windowing functions, can be used in place of cursor code.

There are exceptions to this general guidance to avoid cursors. Some operations need row-at-a time processing or require the results of a cursor to populate variables in dynamic SQL code. For example, database administrators may use cursors to iterate over a list of databases to build a script to perform a maintenance task. However, the exceptions are just that—exceptions to the best practice of avoiding cursors.

Demonstration: Factors in Query Performance

In this demonstration, you will see how to:

- Rewrite a cursor as a set-based query.

Demonstration Steps

Rewrite a Cursor as a Set-based Query

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod19\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod19\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.

MCT USE ONLY. STUDENT USE PROHIBITED

8. Keep SQL Server Management Studio open for the next demonstration.

   Question: For which kinds of SELECT query can SQL Server use indexes to improve performance?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Displaying Query Performance Data

The tools that ship with SQL Server 2014 provide access to a great deal of system-level information about the execution of your queries. In this lesson, you will learn how to use SQL Server Management Studio (SSMS) to display information about query execution plans selected by the SQL Server optimizer, as well as performance data showing elapsed time and resources used by queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the role of execution plans in SQL Server.
- Describe the difference between estimated and actual execution plans.
- Display graphical query execution plans.
- Interpret graphical execution plans.
- Display and interpret basic query execution statistics.

What Is an Execution Plan?

When you submit a query to SQL Server, the database engine goes through several phases in order to execute your code. These include parsing (and checking for syntax errors), resolving (and binding to) object names, and optimizing the query. This last phase produces the execution plan for the query, unless one already exists and was located in SQL Server's cache. The execution plan is then used to execute the query.

- Review of the process of executing a query:
  - Parse, resolve, optimize, execute
- An execution plan includes information on which tables to access, which indexes, what joins to perform
  - If statistics exist for a relevant column or index, then the optimizer will use them in its calculations
- SQL Server tools provide access to execution plans to show how a query was executed or how it would be executed
  - Plans available in text format (deprecated), XML format, and graphical renderings of XML
- Plan viewer accessible in results pane of SSMS

The execution plan contains information about the objects to be accessed by the query, which indexes exist and are relevant, what operators to use, what joins to perform, and estimates about the number of rows involved.

Note: As you learned earlier, the estimated row counts are based on the distribution statistics where available.

SQL Server tools, such as SSMS and some T-SQL commands, provide access to execution plans for queries. These plans, which are available in several formats, provide insight into how SQL Server executes your queries. They can help you and your organization make decisions about how queries are written, which indexes are added, and other tuning techniques.

Plan formats include:

- Plain text, which is being deprecated in the current version of SQL Server.
- XML text, which can be saved, shared, and opened in SSMS or any XML editor.

MCT USE ONLY. STUDENT USE PROHIBITED

- Graphical renderings of XML, which uses icons and other symbols to represent the execution plan data. This lesson will focus on using graphical plans.

Additional reading on execution plans can be found in Books Online at:

Display and Save Execution Plans

http://go.microsoft.com/fwlink/?LinkID=402868

Actual and Estimated Execution Plans

SSMS provides convenient access to graphical execution plans through its menu and toolbar. Two types of plans are accessible—estimated and actual.

Estimated execution plans display how SQL Server would likely execute the query if it were run. Displaying estimated execution plans does not execute the query. Instead, a graphical plan represents the optimizer's estimates. Variances may occur between the estimated plan and the final plan used when the query is run, due to factors such as statistics being out of date.

Instead of producing a graphical plan to generate an XML representation of an estimated execution plan, you can use the SET SHOWPLAN_XML ON T-SQL command, as in the following example:

SET SHOWPLAN_XML

```
USE TSQL;
GO
SET SHOWPLAN_XML ON;
GO
-- Execute a query.
SELECT  custid, ordermonth, qty FROM Sales.CustOrders WHERE custid =4;
GO
SET SHOWPLAN_XML OFF;
```

The results will appear as a hyperlink in the SSMS results pane. Clicking the link will open a new SSMS document window displaying the XML representation of the execution plan. The XML document can be saved. If the document is saved with a .sqlplan extension, SSMS will be associated with it and will interpret it graphically when opened.

The actual execution plan displays the plan that was used by the engine. In SSMS, selecting the option to include the actual plan does not immediately run the query and display the plan. A toggle is set to display the plan with the results when the query is next run within the same SSMS session.

MCT USE ONLY STUDENT USE PROHIBITED

Viewing Graphical Execution Plans

To display the graphical estimated execution plan in SSMS, follow these steps:

1. Enter or select the query text whose plan you wish to view. (As with query execution in SSMS, selecting a portion of text will attempt to produce a plan for the selected portion only. Selecting nothing will operate on all the text in the query window.)
2. On the **Query** menu, click the **Display Estimated Execution Plan** or **Display Estimated Execution Plan** button on the SQL Editor toolbar. Ctrl+L is the default keyboard shortcut.
3. In the **Results** tab, switch to the **Execution Plan** tab.

To include the actual execution plan in SSMS, follow these steps:

1. Enter or select the query text whose plan you wish to view. (As with query execution in SSMS, selecting a portion of text will attempt to produce a plan for the selected portion only. Selecting nothing will operate on all the text in the query window.)
2. On the **Query** menu, click the **Include Actual Execution Plan** or **Include Actual Execution Plan** button on the SQL Editor toolbar. Ctrl+M is the default keyboard shortcut.
3. Execute the query as you would any other query in SSMS: Press F5, click the **Execute Query** toolbar button, or use any other shortcut.
4. In the **Results** tab, switch to the **Execution Plan** tab.

Interpreting the Execution Plan

Execution plans can contain information about a wide range of query operations. This topic is designed to introduce you to some basic operators and concepts to help tune your queries:

- Most execution plans are read from right to left, top to bottom.
- Icons represent the logical and physical operations performed by the query, such as joins, sorts, and index usage. In a SELECT query's plan, the icon representing the SELECT operator itself will typically be the one that is highest and furthest to the left. You might see other operators, depending on the query you submit. The complete list of operators is too long to include in this module, but here are some common ones:

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Operators	Description
Join	Includes nested loops, merges, and hash joins. Indicates SQL Server is matching rows between tables.
Scan and Seek	Indicates SQL Server is retrieving rows from tables and indexes. Scan looks at all rows, whereas seek uses an index to find matches.
Lookup	Includes key and Row ID (RID) lookups. Indicates SQL Server is finding a single row of data.

- Percentages indicate the cost of each operator, relative to the total cost of the plan.
- Arrows show the path of the data through the operators to the final step. The thickness of the arrows' bodies indicates estimated rowcounts in estimated plans and actual rowcounts in actual plans. These lines can draw attention to problem areas within the plan. For example, thick lines leading to a high-cost operator could indicate a bottleneck within the query plan.
- Generally, you won't see differences between the estimated and actual plans, but variances between them might indicate issues with the estimates and, therefore, possible issues with the statistics. Bring variances to the attention of your database administrator.

To learn more about the symbols in an execution plan, see Books Online at:

Showplan Logical and Physical Operators Reference

http://go.microsoft.com/fwlink/?LinkID=402869

Displaying Query Statistics

In addition to execution plan internals, SQL Server can return details about the execution of T-SQL statements by the use of the SET STATISTICS group of commands:

- SQL Server provides detailed runtime information about the execution of a query
- STATISTICS TIME will show time spent parsing and compiling a query

```
SET STATISTICS TIME ON;
```

- STATISTICS IO will show amount of disk activity generated by a query

```
SET STATISTICS IO ON;
```

- SET STATISTICS TIME ON|OFF controls the display of metrics about the time taken to prepare and compile (pre-execution phases), as well as the totals elapsed when the query has completed.
- SET STATISTICS IO ON|OFF controls the display of information about the amount of disk (and data cache) activity generated by a query. Results are displayed in units of 8kb data pages, which store table rows.

For example, the following batch enables STATISTICS TIME, runs a SELECT statement, and disables STATISTICS TIME:

STATISTICS TIME

```
SET STATISTICS TIME ON;
GO
SELECT orderid, custid, empid, orderdate FROM Sales.Orders;
GO
SET STATISTICS TIME OFF;
GO
```

The results look like:

```
SQL Server parse and compile time:
   CPU time = 0 ms, elapsed time = 0 ms.
 (830 row(s) affected)
SQL Server Execution Times:
   CPU time = 0 ms,  elapsed time = 107 ms.
```

These results indicate that the system spent 0 milliseconds preparing the query (it was likely found in plan cache) and 107 milliseconds spent retrieving the results and returning them to the client.

Note: The values depicted are for illustration only. Actual values will depend on many factors, including system performance and caching of plans and data.

The following batch enables STATISTICS IO, runs a SELECT statement, displays the I/O statistics, and then disables STATISTICS IO:

STATISTICS IO

```
SET STATISTICS IO ON;
GO
SELECT orderid, custid, empid, orderdate FROM Sales.Orders;
GO
SET STATISTICS IO OFF;
GO
```

The partial results, edited for clarity:

```
(830 row(s) affected)
Table 'Orders'. Scan count 1, logical reads 21, physical reads 3
```

These results indicate that the system accessed the Sales.Orders table once and read 21 data pages from the data cache and three pages from disk.

For more information about data pages and other on-disk structures in SQL Server, see Books Online at:

Understanding Pages and Extents

http://go.microsoft.com/fwlink/?LinkID=402870

To read more about SET STATISTICS TIME and SET STATISTICS IO, see:

SET STATISTICS TIME (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402871

MCT USE ONLY STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Also see:

SET STATISTICS IO (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402872

Demonstration: Displaying Query Performance Data

In this demonstration, you will see how to:

- Display execution plans and query statistics.

Demonstration Steps

Display Execution Plans and Query Statistics

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod19\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod19\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

   Question: You notice in SSMS that the estimated and actual number of rows affected by a query are significantly different. What action should you take to resolve this problem?

Lab: Improving Query Performance

Scenario

You are a junior database developer for Adventure Works, who has created reports and procedures using corporate databases stored in SQL Server 2014. To determine the performance impact of your queries on the system, you will use SSMS and T-SQL code to monitor your queries as they run in a test environment.

Objectives

After completing this lab, you will be able to:

- View query execution plans.
- View index usage and query statistics.

Estimated Time: 30 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Viewing Query Execution Plans

Scenario

The IT department has supplied T-SQL code to generate a sample table with around 100,000 rows to test different execution plans. The data is based on the Sales.Orders table. You will practice how to observe the execution plan and read basic properties.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Create and populate the sample table Sales.TempOrders
3. Show estimated and actual execution plans
4. Analyze the execution plan of another SELECT statement
5. Graphically compare two execution plans

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.

   Run Setup.cmd in the D:\Labfiles\Lab19\Starter folder as Administrator.

▶ Task 2: Create and populate the sample table Sales.TempOrders

1. In SQL Server Management Studio, open the project file D:\Labfiles\Lab19\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

2. The IT department has provided the following T-SQL code that creates and populates a sample table named Sales.TempOrders:

```
IF OBJECT_ID('Sales.TempOrders') IS NOT NULL
DROP TABLE Sales.TempOrders;
SELECT
orderid, custid, empid, orderdate, requireddate, shippeddate, shipperid, freight,
shipname, shipaddress, shipcity, shipregion, shippostalcode, shipcountry
INTO Sales.TempOrders
FROM Sales.Orders AS o
CROSS JOIN dbo.Nums AS n
WHERE n.n <= 120;
```

3. Execute the provided T-SQL code.

▶ Task 3: Show estimated and actual execution plans

1. Write a SELECT statement to return the orderid, custid, and orderdate columns from the Sales.TempOrders table.
2. Highlight the written statement and show an estimated execution plan. Observe the elements displayed in the execution plan.
3. Hover your mouse pointer over the Table Scan operator in the execution plan and look at the properties displayed in the yellow tooltip box. Notice the word "estimated" in the various properties.
4. Position your mouse pointer over the arrow between the SELECT operator and the Table Scan operator in the execution plan. Which properties are displayed?
5. Show all the properties of the SELECT operator in the execution plan by right-clicking the operator and choosing Properties from the context menu.
6. Click the **Include Actual Execution Plan** button in the SQL Editor toolbar (or press Ctrl+M on the keyboard) and execute the SELECT query.
7. Analyze the actual execution plan. Notice that there are a few new properties with the word "Actual". They represent the actual values gathered during execution.

▶ Task 4: Analyze the execution plan of another SELECT statement

1. Copy the previous SELECT statement and modify it to retrieve only one row using a TOP clause.
2. Show the estimated execution plan.
3. Compare this execution plan with the one in the previous task. Which operator is new?
4. Notice the size of the arrows, as this reflects the number of rows being passed from one operator to another in the execution plan.

▶ Task 5: Graphically compare two execution plans

1. Copy the written SELECT statement in task 2.
2. Copy the modified SELECT statement in task 3, putting it after the first copied SELECT statement.
3. Highlight both statements and first show an estimated plan, and then an actual execution plan.
4. Observe the results. Compare the query cost information provided for the SELECT statements.

Results: After this exercise, you should be able to display estimated and actual execution plans.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Viewing Index Usage and Using SET STATISTICS Statements

Scenario

You will practice how to enable I/O statistics, write two SELECT statements, and observe their execution plans to see how an index was used.

The main tasks for this exercise are as follows:

1. Create a clustered index and write a SELECT statement
2. Enable I/O statistics to observe the number of needed reads
3. Modify the SELECT statement to use a search argument in the WHERE clause
4. Compare both SELECT statements
5. Remove the created table and disable I/O statistics

▶ Task 1: Create a clustered index and write a SELECT statement

1. Open the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. Execute the statement that the IT department provided:

```
CREATE CLUSTERED INDEX CX_Sales_TempOrders_orderdate ON Sales.TempOrders (orderdate
ASC);
```

3. This T-SQL code generates a clustered index on top of the Sales.TempOrders table.
4. After the provided T-SQL code, write a SELECT statement to retrieve the orderid, custid, and orderdate columns from the Sales.TempOrders table. Filter the result to include only the rows with the order year equal to 2007 and the order month equal to 6 by executing the YEAR and MONTH functions against the orderdate column. Execute the SELECT statement.
5. Observe and compare the results that you achieved with the desired results shown in the file 62 - Lab Exercise 2 - Task 1 Result.txt.
6. Show the execution plan. Notice the Clustered Index Scan operator and remember that this is the same operation as a table scan. (Do not worry if some concepts are hard to grasp at this point. They will become clearer when you work more with SQL Server, gain experience, and attend more advanced training courses.)
7. Note that, although you created an index on the orderdate column, the whole table was scanned in search of rows that matched your predicate in the WHERE clause.

▶ Task 2: Enable I/O statistics to observe the number of needed reads

1. Enable I/O statistics by executing the SET STATISTICS IO statement.
2. Copy the written query in task 1 and execute it.
3. Notice the number of logical reads displayed in the **Messages** tab. This number is based on I/O statistics.

▶ Task 3: Modify the SELECT statement to use a search argument in the WHERE clause

1. Copy the written SELECT statement in task 1 and modify it by replacing the existing predicates in the WHERE clause with a range predicate based on the orderdate column. Of course, the result must be the same. Execute the SELECT statement.

2. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab19\Solution\63 - Lab Exercise 2 - Task 3 Result.txt. Notice the number of logical reads displayed in the **Messages** tab. It is more than 25 times lower than the previous SELECT statement.
3. Show the query execution plan. Notice a new operator named Clustered Index Seek.

▶ Task 4: Compare both SELECT statements

1. Copy the written SELECT statement in task 1.
2. Copy the written SELECT statement in task 3, putting it after the first copied SELECT statement.
3. Highlight both statements and execute them.
4. Notice the difference in logical reads. Although the result set is the same, the SELECT statement from task 3 scans 25 times less data than the SELECT statement from task 1.
5. Highlight both statements and show their estimated execution plans. Notice the query cost (relative to batch) between the statements.
6. Why is the SELECT statement from task 3 so much faster?

▶ Task 5: Remove the created table and disable I/O statistics

1. Execute the provided T-SQL code.

Results: After this exercise, you should have a basic understanding of how to enable SET STATISTICS options. Remember to invest time in understanding indexes so that you can write efficient queries.

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways

Review Question(s)

Question: Why should you avoid the use of * in a SELECT clause?

Question: How many clustered indexes are permitted per table?

Question: Which type of execution plan can be displayed without running a query?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 20

Appendix B: Querying SQL Server Metadata

Contents:

Module Overview	**20-1**
Lesson 1: Querying System Catalog Views and Functions	**20-2**
Lesson 2: Executing System Stored Procedures	**20-7**
Lesson 3: Querying Dynamic Management Objects	**20-11**
Lab: Querying SQL Server Metadata	**20-15**
Module Review and Takeaways	**20-20**

Module Overview

In this course, you have learned how to retrieve data from user tables and objects. Whether you plan to write reports or to continue learning Microsoft® SQL Server® development and administration, you will find it useful to query SQL Server for server metadata. When writing queries, it is sometimes necessary to learn which columns are in a particular table, what their data types are, or what collation is used by a string column. You may not have convenient access to a graphical tool such as SQL Server Management Studio (SSMS); however, if you can send queries to SQL Server, you can retrieve the same information that SSMS displays. After all, SSMS is simply issuing metadata queries on your behalf to display its view of a server or a database.

SQL Server provides access to structured metadata through a variety of mechanisms, such as system catalog views, system functions, dynamic management objects, and system stored procedures. In this module, you will learn how to write queries to return system metadata using these mechanisms.

Objectives

After completing this module, you will be able to:

- Write queries that retrieve system metadata using system views and functions.
- Execute system stored procedures to return system information.
- Write queries that retrieve system metadata and state information using system dynamic management views and functions.

Lesson 1
Querying System Catalog Views and Functions

SQL Server provides a common interface to system metadata through a set of views, which you may query using standard T-SQL SELECT statements. By using regular relational views, you may choose which attributes you want to display, join views to form a more complete picture of the system, and set a sort order, just as you would with data stored in user tables.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that retrieve system metadata using system catalog views.
- Write queries that retrieve system metadata using standard information schema views.
- Write queries that retrieve system metadata using system functions.

System Catalog Views

System catalog views contain information about the catalog, or inventory of objects, in a SQL Server system. These views hold a wide range of metadata of interest to developers, administrators, and report writers. This lesson is intended to provide an introduction to system catalog views but is not meant to give a comprehensive listing of all system views.

- Built-in views that provide information about the system catalog
- Use standard query methods to return metadata
  - Column lists, JOIN, WHERE, ORDER BY
- Some views are filtered to display only user objects, some views include system objects

```
--Pre-filtered to exclude system objects
SELECT  name, object_id, schema_id, type, type_desc
FROM sys.tables;
--Includes system and user objects
SELECT name, object_id, schema_id, type, type_desc
FROM sys.objects;
```

In previous versions of SQL Server, Microsoft has tried to consolidate access to system metadata into system catalog views. In the past, users have had to query an assortment of documented and undocumented system tables, system views, and system functions. In the current version of SQL Server, all user-accessible catalog metadata can be found by querying catalog views.

Since you are querying relational views, you may use any of the techniques you have learned in this course to query the system data. For example, the following query retrieves a list of user tables and attributes from the system catalog view sys.tables:

```
USE TSQL;
GO
SELECT  name, object_id, principal_id, schema_id,parent_object_id, type, type_desc
FROM sys.tables;
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The partial results are:

```
name           object_id   principal_id schema_id   parent_object_id type type_desc
-------------- ----------- ------------ ----------- ---------------- ---- ----------
Employees      245575913   NULL         5           0                U    USER_TABLE
Suppliers      309576141   NULL         6           0                U    USER_TABLE
Categories     341576255   NULL         6           0                U    USER_TABLE
Products       373576369   NULL         6           0                U    USER_TABLE
Customers      485576768   NULL         7           0                U    USER_TABLE
Shippers       517576882   NULL         7           0                U    USER_TABLE
Orders         549576996   NULL         7           0                U    USER_TABLE
OrderDetails   645577338   NULL         7           0                U    USER_TABLE
Tests          805577908   NULL         8           0                U    USER_TABLE
Scores         837578022   NULL         8           0                U    USER_TABLE
Nums           901578250   NULL         1           0                U    USER_TABLE
ProductCatalog 1221579390  NULL         7           0                U    USER_TABLE
```

Note that some of the columns reference ID values that are foreign key references to other system tables. Since other system tables are exposed via system views, you can join the sys.tables view to others. For example, the following query joins the sys.tables view with the sys.schemas view to resolve schema IDs into schema names:

```
SELECT  s.name AS schemaname, t.name AS tablename, t.object_id, type_desc,
create_date
FROM sys.tables AS t JOIN sys.schemas AS s ON t.schema_id = s.schema_id
ORDER BY schemaname, tablename;
```

The partial results are:

```
schemaname tablename      object_id   type_desc  create_date
---------- -------------- ----------- ---------- -----------------------
dbo        Nums           901578250   USER_TABLE 2014-02-20 21:07:41.407
dbo        sysdiagrams    1061578820  USER_TABLE 2014-02-20 21:11:43.590
HR         Employees      245575913   USER_TABLE 2014-02-20 21:07:40.070
Production Categories     341576255   USER_TABLE 2014-02-20 21:07:40.093
Production Products       373576369   USER_TABLE 2014-02-20 21:07:40.097
Production Suppliers      309576141   USER_TABLE 2014-02-20 21:07:40.090
Sales      Customers      485576768   USER_TABLE 2014-02-20 21:07:40.100
Sales      OrderDetails   645577338   USER_TABLE 2014-02-20 21:07:40.113
Sales      Orders         549576996   USER_TABLE 2014-02-20 21:07:40.107
Sales      ProductCatalog 1221579390  USER_TABLE 2014-02-20 21:12:54.267
Sales      Shippers       517576882   USER_TABLE 2014-02-20 21:07:40.107
Stats      Scores         837578022   USER_TABLE 2014-02-20 21:07:40.120
Stats      Tests          805577908   USER_TABLE 2014-02-20 21:07:40.117
```

Later in this module, you will also learn how to use a system function to decode the schema name as well as the object_id column.

A list of frequently asked questions (FAQs) about which system catalog object can be used to find certain data is available at:

Querying the SQL Server System Catalog FAQ

http://go.microsoft.com/fwlink/?LinkID=402875

Information Schema Views

- Views stored in the INFORMATION_SCHEMA system schema
- Return system metadata per ISO standard, used by third-party tools
- Maps standard names (catalog, domain) to SQL Server names (database, user-defined data type)

```
SELECT TABLE_CATALOG, TABLE_SCHEMA,
       TABLE_NAME, TABLE_TYPE
FROM   INFORMATION_SCHEMA.TABLES;
```

```
SELECT VIEW_CATALOG, VIEW_SCHEMA, VIEW_NAME,
       TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME,
       COLUMN_NAME
FROM INFORMATION_SCHEMA.VIEW_COLUMN_USAGE;
```

Like system catalog views, information schema views may be queried to return system metadata. Provided to conform with standards, information schema views are useful to third-party tools that may not be written specifically for use with SQL Server. When deciding whether to query SQL Server-specific system views or information schema views, consider the following about the latter:

- They are stored in their own schema, INFORMATION_SCHEMA. SQL Server system views appear in the sys schema.
- They typically use standard terminology instead of SQL Server terms. For example, they use "catalog" instead of "database" and "domain" instead of "user-defined data type." So you need to adjust your queries accordingly.
- They may not expose all the metadata available to SQL Server's own catalog views. For example, sys.columns includes attributes for the identity property and computed column property, while INFORMATION_SCHEMA.columns does not.

To query an information schema view, you use a SELECT statement, as you would with any relational view. For example, the following code queries the INFORMATION_SCHEMA.TABLES view in the TSQL sample database:

```
SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE
FROM INFORMATION_SCHEMA.TABLES;
```

The partial results are:

For more details about information schema views, see Books Online at:

Information Schema Views (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402873

System Metadata Functions

- Return information about settings, values, and objects in SQL Server
- Come in a variety of formats
  - Some marked with a @@ prefix, sometimes incorrectly referred to as global variables: @@VERSION
  - Some marked with a () suffix, similar to arithmetic or string functions: ERROR_NUMBER()
  - Some special functions marked with a $ prefix: $PARTITION

```
SELECT @@VERSION AS SQL_Version;
SELECT SERVERPROPERTY('ProductVersion') AS version;
SELECT SERVERPROPERTY('Collation') AS collation;
```

In addition to views, SQL Server provides a number of built-in functions that return metadata to a query. These include scalar and table-valued functions, which can return information about system settings, session options, and a wide range of objects.

SQL Server metadata functions come in a variety of formats. Some appear similar to standard scalar functions, such as ERROR_NUMBER(). Others use special prefixes, such as @@VERSION or $PARTITION. As you become familiar with and start to use metadata functions, you should consult SQL Server Books Online to determine the format of the functions you need to query.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Some common system metadata functions include:

Function Name	Description	Example
OBJECT_ID(<object_name>)	Returns the object ID of a database object.	OBJECT_ID('Sales.Customer')
OBJECT_NAME(<object_id>)	Returns the name corresponding to an object ID.	OBJECT_NAME(197575742)
@@ERROR	Returns 0 if the last statement succeeded; otherwise returns the error number.	@@ERROR
SERVERPROPERTY(<property >)	Returns the value of the specified server property.	SERVERPROPERTY('Collation')

You use a standard SELECT statement to query a system metadata function. You can include the function in a WHERE clause or in the SELECT clause, as this query shows:

```
SELECT SERVERPROPERTY('EDITION') AS SQLEdition;
```

This query returns:

```
SQLEdition
--------------------------
Developer Edition (64-bit)
```

For more information about SERVERPROPERTY, including the properties it returns, see Books Online at:

SERVERPROPERTY (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402874

Demonstration: Querying System Catalog Views and Functions

In this demonstration, you will see how to:

- Query system catalog views and functions.

Demonstration Steps

Query System Catalog Views and Functions

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod20\Setup.cmd as an administrator.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod20\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Open the **11 – Demonstration A.sql** script file.
7. Follow the instructions contained within the comments of the script file.
8. Keep SQL Server Management Studio open for the next demonstration.

Question: You have been asked to create reports that display system metadata. Your company has databases hosted on SQL Server, Oracle, and MySQL database servers and you have been asked create reports that will work on all these systems if possible. Should you use System Catalog Views, Information Schema Views or System Metadata Functions for these reports?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Executing System Stored Procedures

In addition to views and functions, SQL Server provides system metadata through system stored procedures. In this lesson, you will learn how to execute system stored procedures to return information about the system. First, you will learn the fundamentals needed to execute any stored procedure, and then you will apply that knowledge to carry out system stored procedures.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how SQL Server stored procedures are executed.
- Write queries that retrieve system metadata using system stored procedures.

Executing Stored Procedures

SQL Server exposes metadata through views, functions, and system stored procedures. Earlier in this course, you learned how to access data through views and functions. In this lesson, you will learn how to access stored procedures and return metadata. However, the topic covers only the basics of working with stored procedures. Later in the course, you will learn more about creating basic stored procedures and executing them to retrieve user data.

- Use the EXECUTE or EXEC command before the name of the stored procedure
- Pass parameters by name or position, separated by commas when applicable

```
--no parameters
EXEC sys.sp_databases;

--single parameter
EXEC sys.sp_help N'Sales.Customers';

--multiple named parameters
EXEC sys.sp_tables
        @table_name = '%',
        @table_owner = N'Sales';
```

Like views and user-defined functions, stored procedures are objects whose definitions are stored in a database. However, unlike views and user-defined functions, stored procedures cannot be directly invoked from a SELECT statement or an expression. Instead, you invoke the procedure using the EXECUTE command, providing the name and any parameters if they exist. You may also use EXEC as a shortcut for EXECUTE. For example, to invoke the system metadata procedure sys.sp_databases, you can use the following command:

```
EXEC sys.sp_databases;
```

The sys.sp_databases procedure returns a list of databases on the server, similar to the sys.databases catalog view.

Note that sys.sp_databases does not provide nearly as many attributes as sys.databases. When you are selecting objects to query, you may want to compare the metadata returned by system views to that returned by system procedures. In addition, keep in mind that the output of stored procedures cannot be directly sorted or filtered, nor can columns be selected as they can when querying a view. You may find that system procedures are less flexible than their corresponding system catalog views (when they exist).

Some system stored procedures accept input parameters. To pass parameters to a stored procedure, you supply the parameters in a comma-separated list after the name of the procedure.

For example, to retrieve metadata about a user table, you can call sys.sp_help, which takes the name of the object as a parameter:

sys.sp_help

```
EXEC sys.sp_help @objname = N'Sales.Customers';
```

If there are multiple parameters, you may list them in the order in which they are defined in the procedure's definition, or you may provide named pairs of parameters and values.

For example, the following two queries each provide a valid way of invoking the system stored procedure sys.sp_tables. However, the second method is considered the best practice:

sys.sp_tables

```
EXEC sys.sp_tables 'Customers', 'Sales';
EXEC sys.sp_tables @table_name='Customers', @table_owner='Sales';
```

For more details on executing stored procedures, see the next module or Books Online at:

Execute a Stored Procedure

http://go.microsoft.com/fwlink/?LinkID=402876

Executing System Stored Procedures

Now that you have learned the basics of executing any stored procedure, here are some considerations for using system stored procedures for accessing metadata:

- SQL Server stores the definitions of system stored procedures in a hidden resource database. System procedures appear to be stored in each user and system database, though this is only a logical view. They are not copied into user databases but are accessible from them without specifying the resource database by name.
- Most system stored procedures are marked with an sp_ prefix.
- Always use EXECUTE or its shortcut EXEC to invoke a system stored procedure.
- Include the sys schema name when invoking a system stored procedure.
- Name each parameter and specify its appropriate data type, such as Unicode character string (marked with the N' pref).

For example, if you want to execute a system metadata procedure that returns information about the columns in a table, you would use sys.sp_columns in a query like this:

sys.sp_columns

```
EXEC sys.sp_columns @table_name=N'Customers', @table_owner=N'Sales';
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

Note: As you learned earlier, you cannot directly filter the results of a stored procedure with a WHERE clause, nor can you choose which columns you want to return. Consider using a system catalog view (when a suitable one exists) if you need to customize the results.

Common System Stored Procedures

SQL Server provides numerous system stored procedures, many of which perform administrative tasks. The list of metadata-related procedures includes general help procedures, such as sys.sp_help and sys.sp_helplanguage, as well as more specific ones, such as sys.sp_tables and sys.sp_columns.

- Database engine procedures can provide general metadata
  - sp_help, sp_helplanguage
  - sp_who, sp_lock
- Catalog procedures can be used as an alternative to system catalog views and functions:

Name	Description
sp_databases	Lists databases in an instance of SQL Server
sp_tables	Returns a list of tables or views, except synonyms
sp_columns	Returns column information for the specified objects

- Unlike system views, there is no option to select which columns to return

For example, the sys.sp_helplanguage procedure provides information about date usage in various supported languages. To execute it, you would run the query:

sys.sp_helplanguage

```
EXEC sys.sp_helplanguage;
```

Note: Before trying out new procedures, be sure that you are executing metadata retrieval procedures only, especially if you have administrative or elevated permissions on your SQL Server instance. There is no undo command in T-SQL!

For a list of system catalog stored procedures, see Books Online at:

Catalog Stored Procedures (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402877

Demonstration: Executing System Stored Procedures

In this demonstration, you will see how to:

- Execute system stored procedures.

Demonstration Steps

Execute System Stored Procedures

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod20\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod20\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.

4. Follow the instructions contained within the comments of the script file.
5. Keep SQL Server Management Studio open for the next demonstration.

Question: You have the following query:

```
SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE
FROM sys.sp_table @table_owner=N'Sales'
```

The query generates an error. What mistake have you made?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Querying Dynamic Management Objects

Beginning with SQL Server 2005 and continuing into the current version, Microsoft has tried to consolidate and standardize system metadata and state information into a consistent set of views and functions called dynamic management objects. As with system stored procedures, most of these system views and functions are intended for administrative and development work. However, they can also be used by reports and applications that need to return metadata. In this lesson, you will learn how to use dynamic management objects.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the use of dynamic management objects for returning SQL Server metadata and state information.
- Write queries that retrieve system metadata using dynamic management views and functions.

About Dynamic Management Objects

SQL Server 2014 provides nearly 200 dynamic management objects that return server and database state information. Although dynamic management objects include dynamic management functions, the objects are generally referred to as dynamic management views (DMVs). This avoids confusion with Database Management Objects (DMOs), which is an older SQL Server technology.

- Dynamic management views and functions (DMVs) return server state information
- Nearly 200 DMVs in SQL Server
- DMVs include catalog information as well as administrative status information, such as object dependencies
- DMVs are server-scoped (instance-level) or database-scoped
- Schema name required to invoke
- Requires VIEW SERVER STATE or VIEW DATABASE STATE permission to query DMVs
- Underlying structures change over time, so avoid writing SELECT * queries against DMVs

When planning the use of dynamic management objects in your queries, consider the following guidelines:

- Like system stored procedures, the definitions of DMVs are stored in the hidden resource database and appear logically in each database on a SQL Server instance.
- DMVs are defined with a server scope or database scope, depending on the metadata being accessed. To query DMVs, you must have been granted VIEW SERVER STATE or VIEW DATABASE STATE permission, depending on the scope of the DMV.
- DMVs can be implemented as views or as table-valued functions. If a DMV is a view, it will not take parameters. If it is a function, parameters may be required. Check the documentation for the DMV you wish to use to view its requirements.
- Microsoft has adapted DMVs to address the changing functionality with each new version of SQL Server. Columns are added or dropped to reflect this. Therefore, avoid writing queries that use SELECT * to return all columns from a system DMV to prevent maintenance problems with the code in the future.

Additional reading can be found in Books Online at:

Dynamic Management Views and Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402878

Categorizing DMVs

When looking through the list of DMVs in SSMS or SQL Server Books Online, note that there is a naming convention that helps organize the DMVs by function. After the dm_ prefix, you will see a short descriptive portion of the name. This descriptor places the DMV into a functional category, such as "db" for database-related, "tran" for transaction-related, or "exec" for query execution-related metadata.

- DMVs are categorized by a naming

Naming pattern	Description
db	Database-related information
exec	Query execution-related information
io	I/O statistics
os	SQL Server Operating System (SQLOS) information
tran	Transaction-related information

Within each grouping, you will find one or more DMVs that provide detailed state or metadata information from the system. To query a DMV, you use a SELECT statement, as you would any user-defined view or table-valued function. You may further manipulate the results of the DMV, such as joining it to another DMV, view, or table, or filtering it with a WHERE clause. For examples, see the next topic.

Most of the DMVs require additional knowledge about SQL Server development and administration, which will be covered in Microsoft course 20462D: *Administering Microsoft® SQL Server® Databases*. However, the DMVs themselves are documented in Books Online at:

Dynamic Management Views and Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402878

Querying Dynamic Management Views and Functions

To query a dynamic management object, use a SELECT statement as you would any user-defined view or table-valued function.

For example, the following query returns a list of current user connections from the sys.dm_exec_sessions view:

- Dynamic management views are queried like standard views:

```
SELECT session_id, login_time, program_name
FROM sys.dm_exec_sessions
WHERE is_user_process = 1;
```

- Dynamic management functions are queried as table-valued functions, including parameters:

```
SELECT referencing_schema_name,
       referencing_entity_name,
       referencing_class_desc
FROM   sys.dm_sql_referencing_entities(
       'Sales.Orders', 'OBJECT');
GO
```

sys.dm_exec_sessions

```
SELECT  session_id, login_time,
program_name
FROM sys.dm_exec_sessions
WHERE is_user_process =1;
```

The following code queries sys.dm_sql_referencing_entities, a table-valued dynamic management function that returns metadata about objects whose definitions reference the supplied object. Since this is a function, parameters are supplied:

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

sys.dm_sql_referencing_entities

```
SELECT referencing_schema_name, referencing_entity_name,
       referencing_class_desc
FROM sys.dm_sql_referencing_entities('Sales.Orders', 'OBJECT');
```

When run in the TSQL sample database, the query returns views that depend on the Sales.Orders table, as these results show:

```
referencing_schema_name referencing_entity_name referencing_class_desc
----------------------- ----------------------- ----------------------
Sales                   CustOrders              OBJECT_OR_COLUMN
Sales                   EmpOrders               OBJECT_OR_COLUMN
Sales                   OrdersByEmployeeYear    OBJECT_OR_COLUMN
Sales                   OrderTotalsByYear       OBJECT_OR_COLUMN
Sales                   OrderValues             OBJECT_OR_COLUMN
(5 row(s) affected)
```

Demonstration: Querying Dynamic Management Objects

In this demonstration, you will see how to:

- Query system dynamic management objects.

Demonstration Steps

Query System Dynamic Management Objects

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines, log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod20\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod20\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. Follow the instructions contained within the comments of the script file.
5. Close SQL Server Management Studio without saving any files.

Categorize Activity

Categorize each dynamic management object into the appropriate category. Indicate your answer by writing the category number to the right of each item.

Items	
1	sys.dm_exec_query_optimizer_info
2	sys.dm_db_file_space_usage
3	sys.dm_exec_connections
4	sys.dm_exec_sql_text
5	sys.dm_exec_requests

Items	
6	sys.dm_operation_status
7	sys.dm_db_partition_stats
8	sys.dm_io_pending_io_requests
9	sys.dm_database_copies
10	sys.dm_io_virtual_file_stats
11	sys.dm_io_cluster_valid_path_names

Category 1		Category 2		Category 3
Database Related Information		Query-Execution Related Information		I/O Statistics

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Querying SQL Server Metadata

Scenario

You are a business analyst for Adventure Works, who will be creating reports using corporate databases stored in SQL Server 2014. You need to determine where the data required by your reports is located, and also establish other system characteristics. You will be writing queries against system objects to gather the required metadata.

Objectives

After completing this lab, you will be able to:

- Query system catalog views.
- Query system functions.
- Query system dynamic management views.

Estimated Time: 45 minutes

Virtual machine: **20461C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa$$w0rd**

Exercise 1: Querying System Catalog Views

Scenario

You will practice how to retrieve information about database objects (especially tables and views) and how to get information about columns from the system catalog views.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT statement to retrieve all databases
3. Write a SELECT statement to retrieve all user-defined tables in the TSQL database
4. Use a different approach to retrieve all user-defined tables in the TSQL database
5. Write a SELECT statement to retrieve all columns from the Sales.Customers table

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run Setup.cmd in the D:\Labfiles\Lab20\Starter folder as Administrator.

▶ Task 2: Write a SELECT statement to retrieve all databases

1. In SQL Server Management Studio, open the project file D:\Labfiles\Lab20\Starter\Project\Project.ssmssln and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement that will return the name, dbid, and crdate columns from the view sys.sysdatabases.

3. Execute the SELECT statement and compare the results that you achieved with the desired results shown in the file 52 - Lab Exercise 1 - Task 2 Result.txt. Your result may be different, depending on the databases that exist on the server you are connected to and on your user rights. The desired results were produced by a user with a sysadmin server role.

▶ Task 3: Write a SELECT statement to retrieve all user-defined tables in the TSQL database

1. Write a SELECT statement to retrieve all database objects by selecting the object_id, name, schema_id, type, type_desc, create_date, and modify_date columns from the sys.objects table.
2. Execute the SELECT statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab20\Solution\53 - Lab Exercise 1 - Task 2_1 Result.txt.

   Note: The exact number of rows might differ in your query.
3. Write a SELECT statement to retrieve all the distinct object types by selecting the type and type_desc columns in the sys.objects table. Order the results by the type_desc column.
4. Execute the SELECT statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab20\Solution\54 - Lab Exercise 1 - Task 2_2 Result.txt. In the type_desc column, find a row with the value USER_TABLE. In that row, notice the value in the type column.
5. Copy the first query in this task and modify it to filter only user-based tables.
6. Execute the modified query and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab20\Solution\55 - Lab Exercise 1 - Task 2_3 Result.txt.

   Note: The dates will differ in your query.

▶ Task 4: Use a different approach to retrieve all user-defined tables in the TSQL database

1. Write a SELECT statement against the sys.tables view to show all user-defined tables. Retrieve the same columns as in task 2, but use the system function SCHEMA_NAME with the schema_id column. Use the alias schemaname to display the name of the schema.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab20\Solution\56 - Lab Exercise 1 - Task 3_1 Result.txt.

   Note: The dates are likely to vary from the solution file.
3. Retrieve all views in the TSQL database by writing a SELECT statement against the sys.views view, using the same columns as in the previous query.
4. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab20\Solution\57 - Lab Exercise 1 - Task 3_2 Result.txt.

   Note: The dates are likely to vary from the solution file.

▶ Task 5: Write a SELECT statement to retrieve all columns from the Sales.Customers table

1. Write a SELECT statement to retrieve the columns names from the sys.columns view, using the aliases columnname, column_id, system_type_id, max_length, precision, scale, and collation_name. Filter the results to show only the columns from the Sales.Customers table.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab20\Solution\58 - Lab Exercise 1 - Task 4 Result.txt.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Results: After this exercise, you should be able to retrieve some system information from the system catalog views.

Exercise 2: Querying System Functions

Scenario

You will practice how to use system functions to retrieve additional system information from system catalog views.

The main tasks for this exercise are as follows:

1. Write a SELECT statement to retrieve the current database name

2. Write a SELECT statement to retrieve the object name and schema name

3. Write a SELECT statement to retrieve all the columns from the user-defined tables that contain the word "name" in the column name

4. Retrieve the view definition

▶ Task 1: Write a SELECT statement to retrieve the current database name

1. Open the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve some calculated columns using different system functions:
   - Use the DB_ID function to retrieve the current database identification number. Give this calculated column the alias databaseid.
   - Use the DB_NAME function to retrieve the current database name. Give this calculated column the alias databasename.
   - Use the USER_NAME function to retrieve the current database user name. Give this calculated column the alias currusername.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab20\Solution\62 - Lab Exercise 2 - Task 1 Result.txt. **Note:** The databaseid is likely to vary from the solution file.

▶ Task 2: Write a SELECT statement to retrieve the object name and schema name

1. Write a SELECT statement to retrieve the name column and two calculated columns from the sys.columns view. To retrieve the first calculated column, use the OBJECT_NAME function with the object_id column as a parameter. Give it the alias tablename. To retrieve the second calculated column, use the OBJECT_SCHEMA_NAME function with the object_id column as a parameter. Give it the alias schemaname.
2. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab20\Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

Note: The exact number of rows returned might differ.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Write a SELECT statement to retrieve all the columns from the user-defined tables that contain the word "name" in the column name

1. Write a SELECT statement to retrieve the name column and two calculated columns from the sys.columns view. Give the name column the alias columnname. Use the OBJECT_NAME function to retrieve the first calculated column, giving it the alias tablename. Use the OBJECT_SCHEMA_NAME function to retrieve the second calculated column, giving it the alias schemaname. Filter the result to include only columns that contain the word "name" in the column name and belong to user-defined tables. To do this, use the OBJECTPROPERTY function, passing it two parameters: object_id and IsUserTable with the value 1.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab20\Solution\64 - Lab Exercise 2 – Task 3 Result.txt.

   Note: The results might vary from the solution file.

▶ Task 4: Retrieve the view definition

1. Write a SELECT statement to retrieve the view definition for the Sales.CustOrders view using the OBJECT_DEFINITION function. You will have to pass an object id to the function, so you can use the OBJECT_ID function to get the needed information.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab20\Solution\65 - Lab Exercise 2 - Task 4 Result.txt.

Results: After this exercise, you should know how to use different system functions.

Exercise 3: Querying System Dynamic Management Views

Scenario

To obtain more dynamic information about SQL Server, you will write several SELECT statements to retrieve data from DMVs.

The main tasks for this exercise are as follows:

1. Write a SELECT statement to return all current sessions
2. Execute the provided T-SQL statement
3. Write a SELECT statement to retrieve the current memory information

▶ Task 1: Write a SELECT statement to return all current sessions

1. Open the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the session_id, login_time, host_name, language, and date_format columns from the sys.dm_exec_sessions DMV.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab20\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

   Note: The exact number of rows returned might differ.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Execute the provided T-SQL statement

1. Execute the provided T-SQL statement to retrieve the information about the computer on which SQL Server is installed.

```
SELECT
cpu_count AS 'Logical CPU Count',
hyperthread_ratio AS 'Hyperthread Ratio',
cpu_count/hyperthread_ratio As 'Physical CPU Count',
physical_memory_kb/1024 AS 'Physical Memory (MB)',
sqlserver_start_time AS 'Last SQL Start'
FROM sys.dm_os_sys_info;
```

2. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab20\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

Note: The dates are likely to vary from the solution file

▶ Task 3: Write a SELECT statement to retrieve the current memory information

1. Write a SELECT statement to retrieve the total_physical_memory_kb, available_physical_memory_kb, total_page_file_kb, available_page_file_kb, and system_memory_state_desc columns from the sys.dm_os_sys_memory DMV.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab20\Solution\74 - Lab Exercise 3 - Task 3 Result.txt.

   Note: The results are likely to vary from the solution file.

Results: After this exercise, you should have an understanding of how to write queries against the system DMVs.

Module Review and Takeaways

Review Question(s)

Question: Why might you choose to query a system view rather than a system stored procedure which returned the same metadata?

Question: What issues might you face later if your application used SELECT * to query system catalog views?

MCT USE ONLY. STUDENT USE PROHIBITED

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.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 1: Introduction to Microsoft SQL Server 2014

Lab: Working with SQL Server 2014 Tools

Exercise 1: Working with SQL Server Management Studio

▶ Task 1: Open Microsoft SQL Server Management Studio

1. Start SQL Server Management Studio.
2. In the **Connect to Server window**, click **Cancel**.
3. Close the **Object Explorer** window by clicking the close icon.
4. Close the **Solution Explorer** window by clicking the close icon.
5. Open the **Object Explorer** window by selecting **Object Explorer** on the **View** menu (or press F8 on the keyboard).
6. Open the **Solution Explorer** window by selecting **Solution Explorer** on the **View** menu (or press Ctrl+Alt+L on the keyboard).

▶ Task 2: Configure the Editor Settings

1. On the **Tools** menu, select **Options** to open the **Options** window in SQL Server Management Studio.
2. Expand the **Environment** option and select **Fonts and Colors**. In the **Show settings for** box, select **Text Editor** and set the font size in the **Size** box to **14**.
3. In the left pane, expand the **Text Editor** option, expand the **Transact-SQL** option, and select **IntelliSense**. Under **Transact-SQL IntelliSense Settings**, clear the **Enable IntelliSense** check box.
4. In the left pane, select the **Tabs** option under **Text Editor** and **Transact-SQL**. In the **Tab** frame, change the **Tab size** property to **6**.
5. In the left pane, expand **Query Results**, expand **SQL Server**, and select **Results to Grid**. Enable the option **Include column headers when copying or saving the results** by selecting the check box.
6. Accept the changes by clicking the **OK** button.

Results: After this exercise, you should have opened SSMS and configured editor settings.

Exercise 2: Creating and Organizing T-SQL scripts

▶ Task 1: Create a Project

1. On the **File** menu, select **New** and click **Project**.
2. In the **New Project** window, type **MyFirstProject** in the **Name** text box and **D:\Labfiles\Lab01\Starter** in the **Location** text box. Click the **OK** button to create the new project.
3. In the **Solution Explorer** window, right-click the **Queries** folder under **MyFirstProject** and select **New Query**.
4. In the **Connect to Database Engine Dialog** box, click **Cancel**.
5. Right-click the query file **SQLQuery1.sql** under the **Queries** folder, choose **Rename**, and type **MyFirstQueryFile.sql** as the new name for the file.
6. On the **File** menu, select **Save All**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Add an Additional Query File

1. In the **Solution Explorer** window, right-click the **Queries** folder under **MyFirstProject** and select **New Query**.
2. In the **Connect to Database Engine** Dialog box, click **Cancel**.
3. In the **Queries** folder, right-click the query file **SQLQuery1.sql**, choose **Rename**, and type **MySecondQueryFile.sql** as the new name for the file.
4. Click **File Explorer** on the taskbar.
5. In Windows Explorer, navigate to the folder D:\Labfiles\Lab01\Starter\MyFirstProject\MyFirstProject and observe the created files.
6. In the **Solution Explorer** window in SQL Server Management Studio, right-click the query file **MySecondQueryFile.sql** and select **Remove**. When the confirmation dialog appears, click the **Remove** button.
7. In Windows Explorer, press F5 to refresh the **Windows Explorer** window and notice that the file MySecondQueryFile.sql is still there.
8. In the **Solution Explorer** window in SSMS, right-click the query file **MyFirstQueryFile.sql** and select **Remove**. When the confirmation dialog appears, click the **Delete** button.
9. In Windows Explorer, press F5 to refresh the **Windows Explorer** window. Notice that the file MyFirstQueryFile.sql was deleted from the file system.

▶ Task 3: Reopen the Created Project

1. On the **File** menu, select **Save All**.
2. On the **File** menu, select **Exit** to close the project and SSMS.
3. Click **Start**, click the Down arrow, expand **All Programs**, scroll to the right to the **Microsoft SQL Server 2014** group, and click **SQL Server 2014 Management Studio**.
4. In the **Connect to Server** window, click **Cancel**.
5. On the **File** menu, click **Open** and click **Project / Solution**. In the **Open Project** window, select the project **D:\Labfiles\Lab01\Starter\MyFirstProject\MyFirstProject.ssmssln**.
6. Click **Open**.
7. In Windows Explorer, navigate to the folder D:\Labfiles\Lab01\Starter\MyFirstProject\MyFirstProject. Drag the file **MySecondQueryFile.sql** to the **Queries** folder in the **Solution Explorer** window in SSMS.
8. On the **File** menu, select **Save All**.

Results: After this exercise, you should have a basic understanding of how to create a project in SSMS and add query files to it.

Exercise 3: Using Books Online

▶ Task 1: Launch Books Online

1. On the virtual machine, press the **Windows** key, and type **manage help settings** and click **Manage Help Settings**.
2. In the **Help Library Manager** window, select **Choose online or local help**.
3. Under **Set your preferred help experience**, click **I want to use online help** and click the **OK** button to confirm.
4. Click the **Exit** button to leave the **Help Library Manager** window.

▶ Task 2: Use Books Online

1. On the virtual machine, press the **Windows** key, and type **sql server documentation** and click **SQL Server Documentation**.
2. If Online Help Consent dialog box appears, click **Yes** to continue.
3. In the left pane of the Books Online for SQL Server 2014 website, expand Books Online for SQL Server 2014 and expand Getting Started (SQL Server 2014).
4. Click Documentation for SQL Server 2014 Tools and Add-in Components.
5. Browse the help article.
6. Shut down Internet Explorer.

Results: After this exercise, you should have a basic understanding of how to find information in Books Online.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 2: Introduction to T-SQL Querying

Lab: Introduction to T-SQL Querying

Exercise 1: Executing Basic SELECT Statements

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the **D:\Labfiles\Lab02\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Execute the T-SQL Script

1. On the virtual machine, on the Taskbar, click **SQL Server Management Studio**.
2. In the **Connect to Server** window, in the **Server name** text box, type **MIA-SQL**.
3. Click the **Options** button. Under **Connection Properties**, select **<Browse server>** in the **Connect to database** list. Choose **Yes** when prompted for the connection to the database. Under **User Databases**, select the **TSQL** database. Click **OK**.
4. Click the **Login** tab, select **Windows Authentication** in the **Authentication** list, and click **Connect**.
5. On the File menu, click **Open** and click **Project / Solution**. In the **Open Project** window, select **project D:\Labfiles\Lab02\Starter\Project\Project.ssmssln**.
6. In Solution Explorer, double-click **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard).
7. When the query window opens, click the **Execute** button on the toolbar (or press F5 on the keyboard). You will notice that the TSQL database is selected in the **Available Databases** box. The **Available Databases** box displays the current database context under which the T-SQL script will run. This information is also visible on the status bar.

▶ Task 3: Execute a Part of the T-SQL Script

1. Highlight the following statement under the task 2 description:

```
SELECT
   firstname, lastname, city, country
FROM HR.Employees;
```

   To highlight it, you can move the pointer over the statement while pressing the left mouse button or use the arrow keys to move the pointer while pressing the Shift key.

2. Click **Execute** (or press F5). It is very important to understand that you can highlight a specific part of the code inside the T-SQL script and execute that part alone. If you click Execute without selecting any part of the code, the whole T-SQL script will be executed. If you highlight a specific part of the code by mistake, the SQL Server will attempt to run only that part.

Results: After this exercise, you should know how to open the T-SQL script and execute the whole script or just a specific statement inside it.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Executing Queries That Filter Data Using Predicates

▶ Task 1: Execute the T-SQL Script

1. Close **SQL Server Management Studio**.
2. If prompted to save the files, click **No**.
3. On the virtual machine, on the Taskbar, click **SQL Server 2014 Management Studio**.
4. In the **Connect to Server** window, type **MIA-SQL** in the **Server name** text box.
5. Click the **Options** button. Under **Connection Properties**, select **<Browse server>** in the **Connect to database** list. Choose **Yes** when prompted for the connection to the database. Under **System Databases**, select the **master** database.
6. Click **OK** and click **Connect**.
7. On the **File** menu, click **Open** and click **Project / Solution**. In the **Open Project** window, select project **D:\Labfiles\Lab02\Starter\Project\Project.ssmssln**.
8. In Solution Explorer, double-click **61 - Lab Exercise 2.sql**.
9. When the query window opens, click **Execute**.
10. Notice that you get the error message:

    ```
    Invalid object name 'HR.Employees'.:
    ```

    Why do you think this happened? This error is very common when you are beginning to learn T-SQL. The message tells you that SQL Server could not find the object HR.Employees. This is because the current database context is set to the master database (look at the Available Databases box where the current database is displayed), but the IT department supplied T-SQL scripts to be run against the TSQL database. So, you need to change the database context from master to TSQL. You will learn how to change the database context in the next task.

▶ Task 2: Apply Needed Changes and Execute the T-SQL Script

1. In the **Available Databases** box, select **TSQL** to change the database context.
2. Click **Execute**.
3. Notice that the result from the SELECT statement returns fewer rows than the one in exercise 1. That is because it has a predicate in the WHERE clause to filter out all rows that do not have the value USA in the column country. Only rows for which the logical expression evaluates to TRUE are returned by the WHERE phase to the subsequent logical query processing phase.

▶ Task 3: Uncomment the USE Statement

1. In the script **61 - Lab Exercise 2.sql**, find the line:

    ```
    --USE TSQL;
    ```

2. Delete the first two characters, so that the line looks like this:

    ```
    USE TSQL;
    ```

    By deleting these two characters, you have removed the comment mark. Now the line will not be ignored by SQL Server.

3. On the **File** menu, click **Save 61 - Lab Exercise 2.sql**.

4. On the **File** menu, click **Close**. This will close the T-SQL script.
5. In Solution Explorer, double-click **61 - Lab Exercise 2.sql**.
6. Click **Execute**.
7. Observe the results. Why did the script execute with no errors? The script now includes the uncommented *USE TSQL;* statement. When you execute the whole T-SQL script, the USE statement applies the database context to the TSQL database. The next statement in the T-SQL script then executes against the TSQL database.

Results: After this exercise, you should have a basic understanding of database context and how to change it.

Exercise 3: Executing Queries That Sort Data Using ORDER BY

▶ Task 1: Execute the T-SQL Script

1. In Solution Explorer, double-click **71 - Lab Exercise 3.sql**.
2. Click **Execute**.
3. Notice that the result window is empty. All the statements inside the T-SQL script are commented out, so SQL Server ignores all the statements inside the T-SQL script.

▶ Task 2: Uncomment the Needed T-SQL Statements and Execute Them

1. Locate the line:

```
--USE TSQL;
```

2. Delete the two characters before the USE statement. The line should now look like this:

```
USE TSQL;
```

3. Locate the block comment start element /* after the task 1 description and delete it.
4. Locate the block comment end element */ and delete it. Any text residing within a block starting with /* and ending with */ is treated as a block comment and is ignored by SQL Server.
5. Highlight the statement:

```
USE TSQL;
```

   - Click **Execute**. The database context is now changed to the TSQL database.

6. Highlight the statement:

```
SELECT
firstname, lastname, city, country
FROM HR.Employees
WHERE country = 'USA'
ORDER BY lastname;
```

7. Click **Execute**.
8. Observe the result and notice that the rows are sorted by the lastname column in ascending order.

Results: After this exercise, you should have an understanding of how comments can be specified inside T-SQL scripts.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 3: Writing SELECT Queries

Lab: Writing Basic SELECT Statements

Exercise 1: Writing Simple SELECT Statements

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab03\Starter folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: View all the Tables in the TSQL Database in Object Explorer

1. On the virtual machine, on the Taskbar, click **SQL Server 2014 Management Studio**.
2. In the Connect **to Server** window, in the **Server name** text box, type **MIA-SQL**.
3. Click the **Options** button. Under **Connection Properties**, select **<Browse server>** in the **Connect to database** list. Choose **Yes** when prompted for the connection to the database. Under **User Databases**, select the **TSQL** database. Click **OK**.
4. Click the **Login** tab, select **Windows Authentication** in the Authentication list, and click **Connect**.
5. In Object Explorer, expand the server **MIA-SQL**, expand **Databases**, expand the database **TSQL**, and expand **Tables**.
6. Under **Tables**, notice that there are four table objects in the Sales schema:
   - Sales.Customers
   - Sales.OrderDetails
   - Sales.Orders
   - Sales.Shippers

▶ Task 3: Write a Simple SELECT Statement

1. On the **File** menu, click **Open** and click **Project/Solution**.
2. In the **Open Project** window, open the project **D:\Labfiles\Lab03\Starter\Project\Project.ssmssln**.
3. In Solution Explorer, double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard.)
4. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
5. In the query pane, type the following query after the task 2 description:

```
SELECT    *
FROM Sales.Customers;
```

6. Highlight the query you typed in step 5 and click **Execute**.
7. In the query pane, type the following code after the first query:

```
SELECT    *
FROM
```

MCT USE ONLY. STUDENT USE PROHIBITED

8. In Object Explorer, select the **Sales.Customers** table under **MIA-SQL**, **TSQL**, **Tables**. Using the mouse, drag the selected table into the query pane, after the FROM clause. Then add a semicolon to the end of the SELECT statement.

   Your finished query should look like this:

```
SELECT    *
FROM [Sales].[Customers];
```

9. Highlight the written query and click **Execute**.

▶ Task 4: Write a SELECT Statement that Includes Specific Columns

1. In Object Explorer, expand the **Sales.Customers** table under **MIA-SQL**, **TSQL**, and **Tables**.
2. Expand **Columns** and observe all the columns in the **Sales.Customers** table.
3. In the query pane, type the following query after the task 3 description:

```
SELECT
contactname, address, postalcode, city, country
FROM Sales.Customers;
```

4. Highlight the written query and click **Execute**.
5. Observe the result. How many rows are affected by the last query? There are multiple ways to answer this question using SQL Server Management Studio. One way is to select the previous query and click **Execute**. The total number of rows affected by the executed query is written in the **Results** pane under the **Messages** tab:

```
(91 row(s) affected)
```

   Another way is to look at the status bar displayed below the **Results** pane. On the left side of the status bar, there is a message stating: "Query executed successfully." On the right side, the total number of rows affected by the current query is displayed (91 rows).

Results: After this exercise, you should know how to create simple SELECT statements to analyze existing tables.

Exercise 2: Eliminating Duplicates Using DISTINCT

▶ Task 1: Write a SELECT Statement that Includes a Specific Column

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT    country
FROM Sales.Customers;
```

4. Highlight the written query and click **Execute**.
5. Observe that you have multiple rows with the same values. This occurs because the Sales.Customers table has multiple rows with the same value for the country column.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

▶ Task 2: Write a SELECT Statement that Uses the DISTINCT Clause

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 2 description. On the toolbar, click **Edit** and then **Paste**. You have now copied the previous query to the same query window after the task 2 description.
3. Modify the query by typing **DISTINCT** after the SELECT clause. Your query should look like this:

```
SELECT DISTINCT
   country
FROM Sales.Customers;
```

4. Highlight the written query and click **Execute**.
5. Observe the result and answer these questions:

   How many rows did the query in task 1 return?
   *To answer this question, you can highlight the query written under the task 1 description, click **Execute**, and read the **Results** pane. (If you forgot how to access this pane, look at task 4 in exercise 1.) The number of rows affected by the query is 91.*
   How many rows did the query in Task 2 return?
   *To answer this question, you can highlight the query written under the task 2 description, click **Execute**, and read the **Results** pane. The number of rows affected by the query is 21. This means that there are 21 distinct values for the country column in the Sales.Customers table.*
   Under which circumstances do the following queries against the Sales.Customers table return the same result?

```
SELECT city, region FROM Sales.Customers;
SELECT DISTINCT city, region FROM Sales.Customers;
```

   Both queries would return the same number of rows if all combinations of values in the city and region columns in the Sales.Customers table are unique. If they are not unique, the first query would return more rows than the second one with the DISTINCT clause.
   Is the DISTINCT clause applied to all columns specified in the query or just the first column?
   The DISTINCT clause is always applied to all columns *specified in the SELECT list. It is very important to remember that the DISTINCT clause does not apply to just the first column in the list.*

Results: After this exercise, you should have an understanding of how to return only the different (distinct) rows in the result set of a query.

Exercise 3: Using Table and Column Aliases

▶ Task 1: Write a SELECT Statement that Uses a Table Alias

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT    c.contactname, c.contacttitle
FROM Sales.Customers AS c;
```

Tip: To use the IntelliSense feature when entering column names in a SELECT statement, you can use keyboard shortcuts. To enable IntelliSense, press Ctrl+Q+I. To list all the alias members, position your pointer after the alias and dot (for example, after "c.") and press Ctrl+J.

4. Highlight the written query and click **Execute**.

▶ Task 2: Write A SELECT Statement That Uses Column Aliases

1. In the query pane, type the following query after the task 2 description:

```
SELECT    c.contactname AS Name, c.contacttitle AS Title, c.companyname AS [Company
Name]
FROM Sales.Customers AS c;
```

Observe that the column alias **[Company Name]** is enclosed in square brackets. Column names and aliases that have embedded spaces or reserved keywords must be delimited. This example uses square brackets as the delimiter, but you can also use the ANSI SQL standard delimiter of double quotes, as in "Company Name".

2. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement that Uses a Table Alias and a Column Alias

1. In the query pane, type the following query after the task 3 description:

```
SELECT p.productname AS [Product Name]
FROM Production.Products AS p;
```

2. Highlight the written query and click **Execute**.

▶ Task 4: Analyze and Correct the Query

1. Highlight the written query under the task 4 description and click **Execute**.
2. Observe the result. Note that only one column is retrieved. The problem is that the developer forgot to add a comma after the first column name, so SQL Server treated the second word after the first column name as an alias. For this reason, it is a best practice to always use AS when specifying aliases. That way, it is easier to spot such errors.
3. Correct the query by adding a comma after the first column name. The corrected query should look like this:

```
SELECT city, country
FROM Sales.Customers;
```

Results: After this exercise, you will know how to use aliases for table and column names.

Exercise 4: Using a Simple CASE Expression

▶ Task 1: Write a SELECT Statement

1. In Solution Explorer, double-click the query **81 - Lab Exercise 4.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

3. In the query pane, type the following query after the task 1 description:

```
SELECT p.categoryid, p.productname
FROM Production.Products AS p;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Write a SELECT Statement that Uses a CASE Expression

1. In the query pane, type the following after the task 2 description:

```
SELECT p.categoryid, p.productname,
CASE
WHEN p.categoryid = 1 THEN 'Beverages'
WHEN p.categoryid = 2 THEN 'Condiments'
WHEN p.categoryid = 3 THEN 'Confections'
WHEN p.categoryid = 4 THEN 'Dairy Products'
WHEN p.categoryid = 5 THEN 'Grains/Cereals'
WHEN p.categoryid = 6 THEN 'Meat/Poultry'
WHEN p.categoryid = 7 THEN 'Produce'
WHEN p.categoryid = 8 THEN 'Seafood'
ELSE 'Other'
END AS categoryname
FROM Production.Products AS p;
```

This query uses a CASE expression to add a new column. Note that, when you have a dynamic list of possible values, you usually store them in a separate table. However, for this example, a static list of values is being supplied.

2. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement that Uses a CASE Expression to differentiate Campaign-Focused Products

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 3 description. On the toolbar, click **Edit** and then **Paste**. You have now copied the previous query to the same query window after the task 3 description.
3. Add a new column using an additional CASE expression. Your query should look like this:

```
SELECT    p.categoryid, p.productname,
   CASE
          WHEN p.categoryid = 1 THEN 'Beverages'
          WHEN p.categoryid = 2 THEN 'Condiments'
          WHEN p.categoryid = 3 THEN 'Confections'
          WHEN p.categoryid = 4 THEN 'Dairy Products'
          WHEN p.categoryid = 5 THEN 'Grains/Cereals'
          WHEN p.categoryid = 6 THEN 'Meat/Poultry'
          WHEN p.categoryid = 7 THEN 'Produce'
          WHEN p.categoryid = 8 THEN 'Seafood'
          ELSE 'Other'
   END AS categoryname,
   CASE
          WHEN p.categoryid IN (1, 7, 8) THEN 'Campaign Products'
          ELSE 'Non-Campaign Products'
   END AS iscampaign
FROM Production.Products AS p;
```

4. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

5. In the result, observe that the first CASE expression uses the simple form, whereas the second uses the searched form.

Results: After this exercise, you should know how to use CASE expressions to write simple conditional logic.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 4: Querying Multiple Tables

Lab: Querying Multiple Tables

Exercise 1: Writing Queries That Use Inner Joins

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab04\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Write a SELECT Statement that Uses an Inner Join

1. On the virtual machine, on the Taskbar, click **SQL Server 2014 Management Studio**. In the **Connect to Server** window, type **MIA-SQL** in the **Server name** text box and click **Connect**.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab04\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard).
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
SELECT
p.productname, c.categoryname
FROM Production.Products AS p
INNER JOIN Production.Categories AS c ON p.categoryid = c.categoryid;
```

7. Highlight the written query and click **Execute**.
8. Observe the result and answer these questions:
   - Which column did you specify as a predicate in the ON clause of the join? Why?

     In this query, the categoryid column is the predicate. By intuition, most people would say that it is the predicate because this column exists in both input tables. By the way, using the same name for columns that contain the same data but in different tables is a good practice in data modeling. Another possibility is to check for referential integrity through primary and foreign key information using SQL Server Management Studio. If there are no primary or foreign key constraints, you will have to acquire information about the data model from the developer.
   - Let us say that there is a new row in the Production.Categories table and this new product category does not have any products associated with it in the Production.Products table. Would this row be included in the result of the SELECT statement written under the task 1 description?

     No, because an inner join retrieves only the matching rows based on the predicate from both input tables. Since the new value for the categoryid is not present in the categoryid column in the Production.Products table, there would be no matching rows in the result of the SELECT statement.

Results: After this exercise, you should know how to use an inner join between two tables.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Writing Queries That Use Multiple-Table Inner Joins

▶ Task 1: Execute the T-SQL Statement

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. Highlight the written query under the Task 1 description and click **Execute**.
4. Observe that you get the error message:

```
Ambiguous column name 'custid'.
```

5. This error occurred because the custid column appears in both tables and you have to specify from which table you would like to retrieve the column values.

▶ Task 2: Apply the Needed Changes and Execute the T-SQL Statement

1. Copy the T-SQL statement from Task 1. In the query window, click the line after the Task 2 description. On the toolbar, click **Paste**.
2. Add the column prefix "Customers" to the existing query so that it looks like this:

```
SELECT
Customers.custid, contactname, orderid
FROM Sales.Customers
INNER JOIN Sales.Orders ON Customers.custid = Orders.custid;
```

3. Highlight the modified query and click **Execute**.

▶ Task 3: Change the Table Aliases

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 3 description. On the toolbar, click **Edit** and then **Paste**. You have now copied the previous query to the same query window after the task 3 description.
3. Modify the T-SQL statement to use table aliases. Your query should look like this:

```
SELECT
c.custid, c.contactname, o.orderid
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid;
```

4. Highlight the written query and click **Execute**.
5. Compare the results with the Task 2 results.
6. Modify the T-SQL statement to include a full source table name as the column prefix. Your query should now look like this:

```
SELECT
Customers.custid, Customers.contactname, Orders.orderid
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid;
```

7. Highlight the written query and click **Execute**.

8. Observe that you get the error messages:

```
Msg 4104, Level 16, State 1, Line 2

The multi-part identifier "Customers.custid" could not be found.

Msg 4104, Level 16, State 1, Line 2

The multi-part identifier "Customers.contactname" could not be found.

Msg 4104, Level 16, State 1, Line 2

The multi-part identifier "Orders.orderid" could not be bound.
```

   You received these error messages as, because you are using a different table alias, the full source table name you are referencing as a column prefix no longer exists. Remember that the SELECT clause is evaluated after the FROM clause, so you must use the table aliases when specifying columns in the SELECT clause.

9. Modify the SELECT statement so that it uses the correct table aliases. Your query should look like this:

```
SELECT
c.custid, c.contactname, o.orderid
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid;
```

▶ Task 4: Add an Additional Table and Columns

1. In the query pane, type the following query after the task 4 description:

```
SELECT
c.custid, c.contactname, o.orderid, d.productid, d.qty, d.unitprice
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid;
```

2. Highlight the written query and click **Execute**.
3. Observe the result. Remember that, when you have a multiple-table inner join, the logical query processing is different from the physical implementation. In this case, it means that you cannot guarantee the order in which the SQL Server optimizer will process the tables. For example, you cannot guarantee that the Sales.Customers table will be joined first with the Sales.Orders table, and then with the Sales.OrderDetails table.

Results: After this exercise, you should have a better understanding of why aliases are important and how to do a multiple-table join.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Writing Queries That Use Self Joins

▶ Task 1: Write a Basic SELECT Statement

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid
FROM HR.Employees AS e;
```

4. Highlight the written query and click **Execute**.
5. Observe that the query retrieved nine rows.

▶ Task 2: Write a Query that Uses a Self Join

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 2 description. On the toolbar, click **Edit** and then **Paste**. You have now copied the previous query to the same query window after the task 2 description.
3. Modify the query by adding a self join to get information about the managers. The query should look like this:

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid,
m.lastname AS mgrlastname, m.firstname AS mgrfirstname
FROM HR.Employees AS e
INNER JOIN HR.Employees AS m ON e.mgrid = m.empid;
```

4. Highlight the written query and click **Execute**.
5. Observe that the query retrieved eight rows and answer these questions:
   - Is it mandatory to use table aliases when writing a statement with a self join? Can you use a full source table name as an alias?

     You must use table aliases. You cannot use the full source table name as an alias when referencing both input tables. Eventually, you could use a full source table name as an alias for one input table and another alias for the second input table.
   - Why did you get fewer rows in the result from the T-SQL statement under the task 2 description compared to the result from the T-SQL statement under the task 1 description?

     In task 2's T-SQL statement, the inner join used an ON clause based on manager information (column mgrid). The employee who is the CEO has a missing value in the mgrid column so this row is not included in the result.

Results: After this exercise, you should have an understanding of how to write T-SQL statements that use self joins.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 4: Writing Queries That Use Outer Joins

▶ Task 1: Write a SELECT Statement that Uses an Outer Join

1. In Solution Explorer, double-click the query **81 - Lab Exercise 4.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
c.custid, c.contactname, o.orderid
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON c.custid = o.custid;
```

4. Highlight the written query and click **Execute**.
5. Inspect the result. Notice that the custid 22 and custid 57 rows have a missing value in the orderid column. This is because there are no rows in the Sales.Orders table for these two values of the custid column. In business terms, this means that there are currently no orders for these two customers.

Results: After this exercise, you should have a basic understanding of how to write T-SQL statements that use outer joins.

Exercise 5: Writing Queries That Use Cross Joins

▶ Task 1: Execute the T-SQL Statement

1. In Solution Explorer, double-click the query **91 - Lab Exercise 5.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. Highlight the T-SQL code under the task 1 description and click **Execute**. Don't worry if you do not understand the provided T-SQL code, as it is used here to provide a more realistic example for a cross join in the next task.

▶ Task 2: Write a SELECT Statement that Uses a Cross Join

1. In the query pane, type the following query after the task 2 description:

```
SELECT
e.empid, e.firstname, e.lastname, c.calendardate
FROM HR.Employees AS e
CROSS JOIN HR.Calendar AS c;
```

2. Highlight the written query and click **Execute**.
3. Observe that the query retrieved 3,285 rows and that there are nine rows in the HR.Employees table. Because a cross join produces a Cartesian product of both inputs, it means that there are 365 (3,285/9) rows in the HR.Calendar table.

▶ Task 3: Drop the HR.Calendar Table

1. Highlight the written query under the task 3 description and click **Execute**.

Results: After this exercise, you should have an understanding of how to write T-SQL statements that use cross joins.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 5: Sorting and Filtering Data

Lab: Sorting and Filtering Data

Exercise 1: Writing Queries That Filter Data Using a WHERE Clause

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab05\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Write a SELECT Statement that Uses a WHERE Clause

1. Start SQL Server Management Studio and connect to the MIA-SQL database engine instance using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab05\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard.)
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
SELECT
custid, companyname, contactname, address, city, country, phone
FROM Sales.Customers
WHERE
country = N'Brazil';
```

7. Notice the use of the N prefix for the character literal. This prefix is used because the country column is a Unicode data type. When expressing a Unicode character literal, you need to specify the character N (for National) as a prefix. You will learn more about data types in the next module.
8. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement that Uses an IN Predicate in the WHERE Clause

1. In the query pane, type the following query after the task 2 description:

```
SELECT
custid, companyname, contactname, address, city, country, phone
FROM Sales.Customers
WHERE
country IN (N'Brazil', N'UK', N'USA');
```

2. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Write a SELECT Statement that Uses a LIKE Predicate in the WHERE Clause

1. In the query pane, type the following query after the task 3 description:

```
SELECT
custid, companyname, contactname, address, city, country, phone
FROM Sales.Customers
WHERE
contactname LIKE N'A%';
```

2. Remember that the percent sign (%) wildcard represents a string of any size (including an empty string), whereas the underscore (_) wildcard represents a single character.
3. Highlight the written query and click **Execute**.

▶ Task 5: Observe the T-SQL Statement Provided by the IT Department

1. Highlight the T-SQL statement provided under the task 4 description and click **Execute**.
2. Highlight the provided T-SQL statement. On the toolbar, click **Edit** and then **Copy**.
3. In the query window, click the line after the task 4 description. On the toolbar, click **Edit** and then **Paste**. You have now copied the previous query to the same query window after the task 4 description.
4. Modify the query so that it looks like this:

```
SELECT
c.custid, c.companyname, o.orderid
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON c.custid = o.custid
WHERE
c.city = N'Paris';
```

5. Highlight the modified query and click **Execute**.
6. Observe the result. Is it the same as that of the first SQL statement? The result is not the same. When you specify the predicate in the ON clause, the left outer join preserves all the rows from the left table (Sales.Customers) and adds only the matching rows from the right table (Sales.Orders), based on the predicate in the ON clause. This means that all the customers will show up in the output, but only the ones from Paris will have matching orders. When you specify the predicate in the WHERE clause, the query will filter only the Paris customers. So, be aware that, when you use an outer join, the result of a query in which the predicate is specified in the ON clause can differ from the result of a query in which the predicate is specified in the WHERE clause. (When using an inner join, the results are always the same.) This is because the ON predicate is matching—it defines which rows from the non-preserved side to match to those from the preserved side. The WHERE predicate is a filtering predicate—if a row from either side doesn't satisfy the WHERE predicate, the row is filtered out.

▶ Task 6: Write a SELECT Statement to Retrieve those Customers Without Orders

1. In the query pane, type the following query after the task 5 description:

```
SELECT
c.custid, c.companyname
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON c.custid = o.custid
WHERE o.custid IS NULL;
```

2. It is important to note that, when you are looking for a NULL, you should use the IS NULL, not the equality operator. The equality operator will always return UNKNOWN when you compare something to a NULL. It will even return UNKNOWN when you compare two NULLs.

3. The choice of which attribute to filter from the non-preserved side of the join is also important. You should choose an attribute that can only have a NULL when the row is an outer row (for example, a NULL originating from the base table). For this purpose, three cases are safe to consider:
   - A primary key column. A primary key column cannot be NULL. Therefore, a NULL in such a column can only mean that the row is an outer row.
   - A join column. If a row has a NULL in the join column, it is filtered out by the second phase of the join. So a NULL in such a column can only mean that it is an outer row.
   - A column defined as NOT NULL. A NULL in a column that is defined as NOT NULL can only mean that the row is an outer row.
4. Highlight the written query and click **Execute**.

Results: After this exercise, you should be able to filter rows of data from one or more tables by using WHERE predicates with logical operators.

Exercise 2: Writing Queries That Sort Data Using an ORDER BY Clause

▶ Task 1: Write a SELECT Statement that Uses an ORDER BY Clause

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
c.custid, c.contactname, o.orderid, o.orderdate
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid
WHERE
o.orderdate >= '20080401'
ORDER BY
o.orderdate DESC, c.custid ASC;
```

   Notice the date filter. It uses a literal (constant) of a date. SQL Server recognizes "20080401" as a character string literal and not as a date and time literal, but because the expression involves two operands of different types, one needs to be implicitly converted to the other's type. In this example, the character string literal is converted to the column's data type (DATETIME) because character strings are considered lower in terms of data type precedence, with respect to date and time data types.
   Also notice that the character string literal follows the format "yyyymmdd". Using this format is a best practice because SQL Server knows how to convert it to the correct date, regardless of the language settings.
4. Highlight the written query and click **Execute**.

▶ Task 2: Apply the Needed Changes and Execute the T-SQL Statement

1. Highlight the written query under the task 2 description and click **Execute**.
2. Observe the error message:

```
Invalid column name 'mgrlastname'.
```

MCT USE ONLY. STUDENT USE PROHIBITED

3. This error occurred because the WHERE clause is evaluated before the SELECT clause and, at that time, the column did not have an alias. To fix this problem, you must use the source column name with the appropriate table alias. Modify the T-SQL statement to look like this:

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid,
m.lastname AS mgrlastname, m.firstname AS mgrfirstname
FROM HR.Employees AS e
INNER JOIN HR.Employees AS m ON e.mgrid = m.empid
WHERE
m.lastname = N'Buck';
```

4. Highlight the written query and click **Execute**.

▶ Task 3: Order the Result by the firstname Column

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 3 description. On the toolbar, click **Edit** and then **Paste**. You have now copied the previous query to the same query window after the task 3 description.
3. Modify the T-SQL statement to include an ORDER BY clause that uses the source column name of m.firstname. Your query should look like this:

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid,
m.lastname AS mgrlastname, m.firstname AS mgrfirstname
FROM HR.Employees AS e
INNER JOIN HR.Employees AS m ON e.mgrid = m.empid
ORDER BY
m.firstname;
```

4. Highlight the written query and click **Execute**.
5. Modify the ORDER BY clause so that it uses the alias for the same column (mgrfirstname). Your query should look like this:

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid,
m.lastname AS mgrlastname, m.firstname AS mgrfirstname
FROM HR.Employees AS e
INNER JOIN HR.Employees AS m ON e.mgrid = m.empid
ORDER BY
mgrfirstname;
```

6. Highlight the written query and click **Execute**.
7. Observe the result. Why were you able to use a source column or alias column name? You can use either one because the ORDER BY clause is evaluated after the SELECT clause and the alias for the column name is known.

Results: After this exercise, you should know how to use an ORDER BY clause.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Writing Queries That Filter Data Using the TOP Option

▶ Task 1: Writing Queries That Filter Data Using the TOP Clause

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT TOP (20)
orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Use the OFFSET-FETCH Clause to Implement the Same Task

1. In the query pane, type the following query after the task 2 description:

```
SELECT
orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC
OFFSET 0 ROWS FETCH FIRST 20 ROWS ONLY;
```

2. Remember that the OFFSET-FETCH clause was a new functionality in SQL Server 2012. Unlike the TOP clause, the OFFSET-FETCH clause must be used with the ORDER BY clause.
3. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve the Most Expensive Products

1. In the query pane, type the following query after the task 3 description:

```
SELECT TOP (10) PERCENT
productname, unitprice
FROM Production.Products
ORDER BY unitprice DESC;
```

2. Implementing this task with the OFFSET-FETCH clause is possible but not easy because, unlike TOP, OFFSET-FETCH does not support a PERCENT option.
3. Highlight the written query and click **Execute**.

Results: After this exercise, you should have an understanding of how to apply the TOP option in the SELECT clause of a T-SQL statement.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 4: Writing Queries That Filter Data Using the OFFSET-FETCH Clause

▶ Task 1: OFFSET-FETCH Clause to Fetch the First 20 Rows

1. In Solution Explorer, double-click the query **81 - Lab Exercise 4.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
custid, orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate, orderid
OFFSET 0 ROWS FETCH FIRST 20 ROWS ONLY;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Use the OFFSET-FETCH Clause to Skip the First 20 Rows

1. In the query pane, type the following query after the task 2 description:

```
SELECT
custid, orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate, orderid
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Write a Generic Form of the OFFSET-FETCH Clause for Paging

1. Solution: OFFSET (@pagenum - 1) * @pagesize ROWS FETCH NEXT @pagesize ROWS ONLY.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 6: Working with SQL Server 2014 Data Types

Lab: Working with SQL Server 2014 Data Types

Exercise 1: Writing Queries That Return Date and Time Data

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab06\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish

▶ Task 2: Write a SELECT Statement to Retrieve all Distinct Customers

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab06\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard).
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
SELECT
CURRENT_TIMESTAMP AS currentdatetime,
CAST(CURRENT_TIMESTAMP AS DATE) AS currentdate,
CAST(CURRENT_TIMESTAMP AS TIME) AS currenttime,
YEAR(CURRENT_TIMESTAMP) AS currentyear,
MONTH(CURRENT_TIMESTAMP) AS currentmonth,
DAY(CURRENT_TIMESTAMP) AS currentday,
DATEPART(week, CURRENT_TIMESTAMP) AS currentweeknumber,
DATENAME(month, CURRENT_TIMESTAMP) AS currentmonthname;
```

   This query uses the CURRENT_TIMESTAMP function to return the current date and time. You can also use the SYSDATETIME function to get a more precise time element compared to the CURRENT_TIMESTAMP function.
   Note that you cannot use the alias currentdatetime as the source in the second column calculation because SQL Server supports a concept called all-at-once operations. This means that all expressions appearing in the same logical query processing phase are evaluated as if they occurred at the same point in time. This concept explains why, for example, you cannot refer to column aliases assigned in the SELECT clause within the same SELECT clause, even if it seems intuitive that you should be able to.

7. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Write a SELECT Statement to Return the Data Type date

1. In the query pane, type the following queries after the task 2 description. The first query uses SQL Server 2014's new DATEFROMPARTS function:

```
SELECT DATEFROMPARTS(2011, 12, 11) AS somedate;
SELECT CAST('20111211' AS DATE) AS somedate;
SELECT CONVERT(DATE, '12/11/2011', 101) AS somedate;
```

2. Highlight the written queries and click **Execute**.

▶ Task 4: Write a SELECT Statement that Uses Different Date and Time Functions

1. In the query pane, type the following query after the task 3 description:

```
SELECT
DATEADD(month, 3, CURRENT_TIMESTAMP) AS threemonths,
DATEDIFF(day, CURRENT_TIMESTAMP, DATEADD(month, 3, CURRENT_TIMESTAMP)) AS diffdays,
DATEDIFF(week, '19920404', '20110916') AS diffweeks,
DATEADD(day, -DAY(CURRENT_TIMESTAMP) + 1, CURRENT_TIMESTAMP) AS firstday;
```

2. Highlight the written query and click **Execute**.

▶ Task 5: Observe the Table Provided by the IT Department

1. Highlight the written query under the task 4 description and click **Execute**.
2. In the query pane, type the following queries after the task 4 description:

```
SELECT
isitdate,
CASE WHEN ISDATE(isitdate) = 1 THEN CONVERT(DATE, isitdate) ELSE NULL END AS
converteddate
FROM Sales.Somedates;
--Uses the TRY_CONVERT function:
SELECT
isitdate,
TRY_CONVERT(DATE, isitdate) AS converteddate
FROM Sales.Somedates;
```

   The second query uses the TRY_CONVERT function. This function returns a value casted to the specified data type if the casting succeeds; otherwise, it returns NULL. Do not worry if you do not recognize the type conversion functions as they will be covered in the next module.

3. Highlight the written queries and click **Execute**.
4. Observe the result and answer these questions:
   - What is the difference between the SYSDATETIME and CURRENT_TIMESTAMP functions?

     There are two main differences. First, the SYSDATETIME function provides a more precise time element compared to the CURRENT_TIMESTAMP function. Second, the SYSDATETIME function returns the data type datetime2(7), whereas the CURRENT_TIMESTAMP returns the data type datetime.

   - What is a language-neutral format for the data type date?

     You can use the format 'YYYYMMDD' or 'YYYY-MM-DD'.

Results: After this exercise, you should be able to retrieve date and time data using T-SQL.

Exercise 2: Writing Queries That Use Date and Time Functions

▶ Task 1: Write a SELECT Statement to Retrieve All Distinct Customers

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT DISTINCT
custid
FROM Sales.Orders
WHERE
YEAR(orderdate) = 2008
AND MONTH(orderdate) = 2;
```

4. Highlight the written query and click **Execute**.
5. Note that you could also write a query that uses a range format, which would better utilize indexing. The query would then look like this:

```
SELECT DISTINCT
custid
FROM Sales.Orders
WHERE
orderdate >= '20080201'
AND orderdate < '20080301';
```

▶ Task 2: Write a SELECT Statement to Calculate the First and Last Day of the Month

1. In the query pane, type the following query after the task 2 description:

```
SELECT
CURRENT_TIMESTAMP AS currentdate,
DATEADD (day, 1, EOMONTH(CURRENT_TIMESTAMP, -1)) AS firstofmonth,
EOMONTH(CURRENT_TIMESTAMP) AS endofmonth;
```

2. This query uses the EOMONTH function, which was new in SQL Server 2012.
3. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve the Orders Placed in the Last Five Days of the Ordered Month

1. In the query pane, type the following query after the task 3 description:

```
SELECT
orderid, custid, orderdate
FROM Sales.Orders
WHERE
DATEDIFF(
day,
orderdate,
EOMONTH(orderdate)
) < 5;
```

2. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Write a SELECT Statement to Retrieve All Distinct Products Sold in the First 10 Weeks of the Year 2007

1. In the query pane, type the following query after the task 4 description:

```
SELECT DISTINCT
d.productid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE
DATEPART(week, orderdate) <= 10
AND YEAR(orderdate) = 2007;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should know how to use the date and time functions.

Exercise 3: Writing Queries That Return Character Data

▶ Task 1: Write a SELECT Statement to Concatenate Two Columns

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
contactname + N' (city: ' + city + N')' AS contactwithcity
FROM Sales.Customers;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Add an Additional Column and Treat NULL as an Empty String

You will have to use the COALLESCE function which will be covered in module 8. You can either look at the solution or skip this task.

1. In the query pane, type the following query after the task 2 description:

```
SELECT
contactname + N' (city: ' + city + N', region: ' + COALESCE(region, '') + N')' AS
fullcontact
FROM Sales.Customers;
```

This query uses the COALESCE function to replace a NULL with an empty string. The next module will include more examples of how to handle a NULL.

2. Highlight the written query and click **Execute**.
3. Note that you can also use the CONCAT function to concatenate strings. It also replaces a NULL with an empty string. The query using the CONCAT function would look like this:

```
SELECT
CONCAT(contactname, N' (city: ', city,  N', region: ', region, N')') AS fullcontact
FROM Sales.Customers;
```

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Write a SELECT Statement to Retrieve All Customers Based on the First Character in the Contact Name

1. In the query pane, type the following query after the task 3 description:

```
SELECT contactname, contacttitle
FROM Sales.Customers
WHERE contactname LIKE N'[A-G]%'
ORDER BY contactname;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should have an understanding of how to concatenate character data.

Exercise 4: Writing Queries That Use Character Functions

▶ Task 1: Write a SELECT Statement that Uses the SUBSTRING Function

1. In Solution Explorer, double-click the query **81 - Lab Exercise 4.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
contactname,
SUBSTRING(contactname, 0, CHARINDEX(N',', contactname)) AS lastname
FROM Sales.Customers;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Extend the SUBSTRING Function to Retrieve the First Name

1. In the query pane, type the following query after the task 2 description:

```
SELECT
REPLACE(contactname, ',', '') AS newcontactname,
SUBSTRING(contactname, CHARINDEX(N',', contactname)+1, LEN(contactname)-
CHARINDEX(N',', contactname)+1) AS firstname
FROM Sales.Customers;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Change the Customer IDs

1. In the query pane, type the following query after the task 3 description:

```
SELECT
custid,
N'C' + RIGHT(REPLICATE('0', 5) + CAST(custid AS VARCHAR(5)), 5) AS custnewid
FROM Sales.Customers;
```

2. Highlight the written query and click **Execute**.
3. Note that you can also use the FORMAT function. The query would then look like this:

```
SELECT
FORMAT(custid, N'\C00000')
FROM Sales.Customers;
```

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Challenge: Write a SELECT Statement to Return the Number of Character Occurrences

1. In the query pane, type the following query after the task 4 description:

```
SELECT
contactname,
LEN(contactname) - LEN(REPLACE(contactname, 'a', '')) AS numberofa
FROM Sales.Customers
ORDER BY numberofa DESC;
```

   This elegant solution first returns the number of characters in the contact name, and then subtracts the number of characters in the contact name without the character 'a'. The result is stored in a new column named numberofa.

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should have an understanding of how to use the character functions.

MCT USE ONLY STUDENT USE PROHIBITED

Module 7: Using DML to Modify Data

Lab: Using DML to Modify Data

Exercise 1: Inserting Records with DML

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab07\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Insert a Row

1. Start SQL Server Management Studio and connect to the MIA-SQL database engine using Windows authentication.
2. In the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab07\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **41 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard.)
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
INSERT INTO HR.Employees(Title, titleofcourtesy,FirstName, Lastname,hiredate,
birthdate, address, city, country, phone)
VALUES
('Sales Representative',
'Mr',
'Laurence',
'Grider',
'04/04/2013',
'10/25/1975',
'1234 1st Ave. S.E. ',
'Seattle',
'USA',
'(206)555-0105');
```

7. Click **Execute**.

▶ Task 3: Insert a Row with SELECT

1. Click **New Query**.
2. In the query pane, type the following query:

```
INSERT INTO Sales.Customers(
companyname,
contactname,
contacttitle,
address,
city,
region,
postalcode,
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
country,
phone,
fax)
SELECT * FROM dbo.PotentialCustomers;
```

3. Click **Execute**.

Results: After successfully completing this exercise, you will have one new employee and three new customers.

Exercise 2: Update and Delete Records Using DML

▶ Task 1: Update Rows

1. Click **New Query**.
2. In the query pane, type the following query:

```
UPDATE Sales.Customers
SET contacttitle='Sales Consultant'
WHERE city='Berlin'
AND contacttitle='Sales Representative';
```

3. Click **Execute**.

▶ Task 2: Delete Rows

1. Click **New Query**.
2. In the query pane, type the following query:

```
DELETE FROM dbo.PotentialCustomers
WHERE contactname IN(
'Taylor, Maurice',
'Mallit, Ken',
'Tiano, Mike');
```

3. Click **Execute**.

Results: After successfully completing this exercise, you will have updated all the records in the Customers table which have a city of Berlin and a contacttitle of Sales Representative to have a contacttitle of Sales Consultant. You will also have deleted the three records in the PotentialCustomers table, which have already been added to the Customers table.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 8: Using Built-In Functions

Lab: Using Built-In Functions

Exercise 1: Writing Queries That Use Conversion Functions

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab08\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Write a SELECT Statement that Uses the CAST or CONVERT Function

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab08\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard.)
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
SELECT N'The unit price for the ' + productname + N' is ' + CAST(unitprice AS
NVARCHAR(10)) + N' $.' AS productdesc
FROM Production.Products;
```

   This query uses the CAST function rather than the CONVERT function. It is better to use the CAST function because it is an ANSI SQL standard. You should use the CONVERT function only when you need to apply a specific style during a conversion.

7. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Filter Rows Based on Specific Date Information

1. In the query pane, type the following query after the task 2 description:

```
SELECT orderid, orderdate, shippeddate, COALESCE(shipregion, 'No region') AS
shipregion
FROM Sales.Orders
WHERE
orderdate >= CONVERT(DATETIME, '4/1/2007', 101)
AND orderdate <= CONVERT(DATETIME, '11/30/2007', 101)
AND shippeddate > DATEADD(DAY, 30, orderdate);
```

2. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

3. Note that you could also write a solution using the PARSE function. The query would look like this:

```
SELECT orderid, orderdate, shippeddate, COALESCE(shipregion, 'No region') AS
shipregion
FROM Sales.Orders
WHERE
orderdate >= PARSE('4/1/2007' AS DATETIME USING 'en-US')
AND orderdate <= PARSE('11/30/2007' AS DATETIME USING 'en-US')
AND shippeddate > DATEADD(DAY, 30, orderdate);
```

▶ Task 4: Write a SELECT Statement to Convert the Phone Number Information to an Integer Value

1. In the query pane, type the following query after the task 3 description:

```
SELECT
CONVERT(INT, REPLACE(REPLACE(REPLACE(REPLACE(phone, N'-', N''), N'(', ''), N')', ''),
' ', '')) AS phonenoasint
FROM Sales.Customers;
```

   This query is trying to use the CONVERT function to convert phone numbers that include characters such as hyphens and parentheses into an integer value.

2. Highlight the written query and click **Execute**.
3. Observe the error message:

```
Conversion failed when converting the nvarchar value '67.89.01.23' to data type int.
```

   Because you want to retrieve rows without conversion errors and have a NULL for those that produce a conversion error, you can use the TRY_CONVERT function.

4. Modify the query to use the TRY_CONVERT function. The query should look like this:

```
SELECT
TRY_CONVERT(INT, REPLACE(REPLACE(REPLACE(REPLACE(phone, N'-', N''), N'(', ''), N')',
''), ' ', '')) AS phonenoasint
FROM Sales.Customers;
```

5. Highlight the written query and click **Execute**. Observe the result. The rows that could not be converted have a NULL.

Results: The unit price for the Product HHYDP is 18.00 $.

Exercise 2: Writing Queries That Use Logical Functions

▶ Task 1: Write a SELECT Statement to Mark Specific Customers Based on their Country and Contact Title

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
IIF(country = N'Mexico' AND contacttitle = N'Owner', N'Target group', N'Other') AS
segmentgroup, custid, contactname
FROM Sales.Customers;
```

MCT USE ONLY STUDENT USE PROHIBITED

4. The IIF function was new in SQL Server 2012. It was added mainly to support migrations from Microsoft Access to SQL Server. You can always use a CASE expression to achieve the same result.
5. Highlight the written query and click **Execute**.

▶ Task 2: Modify the T-SQL Statement to Mark Different Customers

1. In the query pane, type the following query after the task 2 description:

```
SELECT
IIF(contacttitle = N'Owner' OR region IS NOT NULL, N'Target group', N'Other') AS
segmentgroup,  custid,  contactname
FROM Sales.Customers;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Create Four Groups of Customers

1. In the query pane, type the following query after the task 3 description:

```
SELECT CHOOSE(custid % 4 + 1, N'Group One', N'Group Two', N'Group Three', N'Group
Four') AS segmentgroup, custid, contactname
FROM Sales.Customers;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should know how to use the logical functions.

Exercise 3: Writing Queries That Test for Nullability

▶ Task 1: Write a SELECT Statement to Retrieve the Customer Fax Information

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT contactname, COALESCE(fax, N'No information') AS faxinformation
FROM Sales.Customers;
```

This query uses the COALESCE function to retrieve customers' fax information.

4. Highlight the written query and click **Execute**.
5. In the query pane, type the following query after the previous query:

```
SELECT contactname, ISNULL(fax, N'No information') AS faxinformation
FROM Sales.Customers;
```

This query uses the ISNULL function. What is the difference between the ISNULL and COALESCE functions? COALESCE is a standard ANSI SQL function and ISNULL is not. So you should use the COALESCE function.

6. Highlight the written query and click **Execute**.

▶ Task 2: Write a Filter for a Variable that Could Be a Null

1. Highlight the query provided under the task 2 description and click **Execute**.
2. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.

MCT USE ONLY. STUDENT USE PROHIBITED

3. In the query window, click the line after the task 2 description. On the toolbar, click **Edit** and then **Paste**. You have now copied the previous query to the same query window after the task 2 description.
4. Modify the query so that it looks like this:

```
DECLARE @region AS NVARCHAR(30) = NULL;
SELECT
custid, region
FROM Sales.Customers
WHERE region = @region OR (region IS NULL AND @region IS NULL);
```

5. Highlight the modified query and click **Execute**.
6. Test the modified query by setting the @region parameter to N'WA'. The T-SQL expression should look like this:

```
DECLARE @region AS NVARCHAR(30) = N'WA';
SELECT
custid, region
FROM Sales.Customers
WHERE region = @region OR (region IS NULL AND @region IS NULL);
```

7. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Return All the Customers that Do Not Have a Two-Character Abbreviation for the Region

1. In the query pane, type the following query after the task 3 description:

```
SELECT custid, contactname, city, region
FROM Sales.Customers
WHERE
region IS NULL
OR LEN(region) <> 2;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should have an understanding of how to test for nullability.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 9: Grouping and Aggregating Data

Lab: Grouping and Aggregating Data

Exercise 1: Writing Queries That Use the GROUP BY Clause

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab09\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Write a SELECT Statement to Retrieve Different Groups of Customers

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab09\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard).
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
SELECT
o.custid, c.contactname
FROM Sales.Orders AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE o.empid = 5
GROUP BY o.custid, c.contactname;
```

7. Highlight the written query and click **Execute**.

▶ Task 3: Add an Additional Column From the Sales.Customers Table

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 2 description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the T-SQL statement so that it adds an additional column. Your query should look like this:

```
SELECT
o.custid, c.contactname, c.city
FROM Sales.Orders AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE o.empid = 5
GROUP BY o.custid, c.contactname;
```

4. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

5. Observe the error message:

   Column 'Sales.Customers.city' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

   Why did the query fail? In a grouped query, you will get an error if you refer to an attribute that is not in the GROUP BY list (such as the city column) or not an input to an aggregate function in any clause that is processed after the GROUP BY clause.

6. Modify the SQL statement to include the city column in the GROUP BY clause. Your query should look like this:

```
SELECT
o.custid, c.contactname, c.city
FROM Sales.Orders AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE o.empid = 5
GROUP BY o.custid, c.contactname, c.city;
```

7. Highlight the written query and click **Execute**.

▶ Task 4: Write a SELECT Statement to Retrieve the Customers with Orders for Each Year

1. In the query pane, type the following query after the task 3 description:

```
SELECT
custid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE empid = 5
GROUP BY custid, YEAR(orderdate)
ORDER BY custid, orderyear;
```

2. Highlight the written query and click **Execute**.

▶ Task 5: Write a SELECT Statement to Retrieve Groups of Product Categories Sold in a Specific Year

1. In the query pane, type the following query after the task 4 description:

```
SELECT
c.categoryid, c.categoryname
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
INNER JOIN Production.Products AS p ON p.productid = d.productid
INNER JOIN Production.Categories AS c ON c.categoryid = p.categoryid
WHERE orderdate >= '20080101' AND orderdate < '20090101'
GROUP BY c.categoryid, c.categoryname;
```

2. Highlight the written query and click **Execute**.

   Important note regarding the use of the DISTINCT clause:
   In all the tasks in Exercise 1, you could use the DISTINCT clause in the SELECT clause as an alternative to using a grouped query. This is possible because aggregate functions are not being requested.

Results: After this exercise, you should be able to use the GROUP BY clause in the T-SQL statement.

MCT USE ONLY, STUDENT USE PROHIBITED

Exercise 2: Writing Queries That Use Aggregate Functions

▶ Task 1: Write a SELECT Statement to Retrieve the Total Sales Amount Per Order

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
o.orderid, o.orderdate, SUM(d.qty * d.unitprice) AS salesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.orderid, o.orderdate
ORDER BY salesamount DESC;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Add Additional Columns

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 2 description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the T-SQL statement so that it adds additional columns. Your query should look like this:

```
SELECT
o.orderid, o.orderdate,
SUM(d.qty * d.unitprice) AS salesamount,
COUNT(*) AS noofoderlines,
AVG(d.qty * d.unitprice) AS avgsalesamountperorderline
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.orderid, o.orderdate
ORDER BY salesamount DESC;
```

4. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve the Sales Amount Value Per Month

1. In the query pane, type the following query after the task 3 description:

```
SELECT
YEAR(orderdate) * 100 + MONTH(orderdate) AS yearmonthno,
SUM(d.qty * d.unitprice) AS saleamountpermonth
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY YEAR(orderdate), MONTH(orderdate)
ORDER BY yearmonthno;
```

2. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Write a SELECT Statement to List All Customers with the Total Sales Amount and Number of Order Lines Added

1. In the query pane, type the following query after the task 4 description:

```
SELECT
c.custid, c.contactname,
SUM(d.qty * d.unitprice) AS totalsalesamount,
MAX(d.qty * d.unitprice) AS maxsalesamountperorderline,
COUNT(*) AS numberofrows,
COUNT(o.orderid) AS numberoforderlines
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON o.custid = c.custid
LEFT OUTER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY c.custid, c.contactname
ORDER BY totalsalesamount;
```

2. Highlight the written query and click **Execute**.
3. Observe the result. Notice that the values in the numberofrows and numberoforderlines columns are different. Why? All aggregate functions ignore NULLs except COUNT(*), which is why you received the value 1 for the numberofrows column. When you used the orderid column in the COUNT function, you received the value 0 because the orderid is NULL for customers without an order.

Exercise 3: Writing Queries That Use Distinct Aggregate Functions

▶ Task 1: Modify a SELECT Statement to Retrieve the Number of Customers

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. Highlight the provided T-SQL statement after the Task 1 description and click **Execute**.
4. Observe the result. Notice that the number of orders is the same as the number of customers. Why? You are using the aggregate COUNT function on the orderid and custid columns and, since every order has a customer, the COUNT function returns the same value. It does not matter if there are multiple orders for the same customer because you are not using a DISTINCT clause inside the aggregate function. If you want to get the correct number of distinct customers, you have to modify the provided T-SQL statement to include a DISTINCT clause.
5. Modify the provided T-SQL statement to include a DISTINCT clause. The query should look like this:

```
SELECT
YEAR(orderdate) AS orderyear,
COUNT(orderid) AS nooforders,
COUNT(DISTINCT custid) AS noofcustomers
FROM Sales.Orders
GROUP BY YEAR(orderdate);
```

6. Highlight the written query and click **Execute**.

▶ Task 2: Write a SELECT Statement to Analyze Segments of Customers

1. In the query pane, type the following query after the task 2 description:

```
SELECT
   SUBSTRING(c.contactname,1,1) AS firstletter,
   COUNT(DISTINCT c.custid) AS noofcustomers,
   COUNT(o.orderid) AS nooforders
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON o.custid = c.custid
GROUP BY SUBSTRING(c.contactname,1,1)
ORDER BY firstletter;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve Additional Sales Statistics

1. In the query pane, type the following query after the task 3 description:

```
SELECT
c.categoryid, c.categoryname,
SUM(d.qty * d.unitprice) AS totalsalesamount,
COUNT(DISTINCT o.orderid) AS nooforders,
SUM(d.qty * d.unitprice) / COUNT(DISTINCT o.orderid) AS avgsalesamountperorder
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
INNER JOIN Production.Products AS p ON p.productid = d.productid
INNER JOIN Production.Categories AS c ON c.categoryid = p.categoryid
WHERE orderdate >= '20080101' AND orderdate < '20090101'
GROUP BY c.categoryid, c.categoryname;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should have an understanding of how to apply a DISTINCT aggregate function.

Exercise 4: Writing Queries That Filter Groups with the HAVING Clause

▶ Task 1: Write a SELECT Statement to Retrieve the Top 10 Customers

1. In Solution Explorer, double-click the query **81 - Lab Exercise 4.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT TOP (10)
o.custid,
SUM(d.qty * d.unitprice) AS totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING SUM(d.qty * d.unitprice) > 10000
ORDER BY totalsalesamount DESC;
```

4. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Write a SELECT Statement to Retrieve Specific Orders

1. In the query pane, type the following query after the task 2 description:

```
SELECT
o.orderid,
o.empid,
SUM(d.qty * d.unitprice) as totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.orderdate >= '20080101' AND o.orderdate < '20090101'
GROUP BY o.orderid, o.empid;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Apply Additional Filtering

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 3 description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the T-SQL statement to apply additional filtering. Your query should look like this:

```
SELECT
o.orderid,
o.empid,
SUM(d.qty * d.unitprice) as totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.orderdate >= '20080101' AND o.orderdate < '20090101'
GROUP BY o.orderid, o.empid
HAVING SUM(d.qty * d.unitprice) >= 10000;
```

4. Highlight the written query and click **Execute**.
5. Modify the T-SQL statement to include an additional filter to retrieve only orders handled by the employee whose ID is 3. Your query should look like this:

```
SELECT
o.orderid,
o.empid,
SUM(d.qty * d.unitprice) as totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE
o.orderdate >= '20080101' AND o.orderdate <= '20090101'
AND o.empid = 3
GROUP BY o.orderid, o.empid
HAVING SUM(d.qty * d.unitprice) >= 10000;
```

   In this query, the predicate logic is applied in the WHERE clause. You could also write the predicate logic inside the HAVING clause. Which do you think is better? Unlike with orderdate filtering, with empid filtering, the result is going to be correct either way because you are filtering by an element that appears in the GROUP BY list. Conceptually, it seems more intuitive to filter as early as possible. This query then applies the filtering in the WHERE clause because it will be logically applied before the GROUP BY clause. Do not forget, though, that the actual processing in the SQL Server engine could be different.

6. Highlight the written query and click **Execute**.

▶ Task 4: Retrieve the Customers with More Than 25 Orders

1. In the query pane, type the following query after the task 4 description:

```
SELECT
o.custid,
MAX(orderdate) AS lastorderdate,
SUM(d.qty * d.unitprice) AS totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING COUNT(DISTINCT o.orderid) > 25;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should have an understanding of how to use the HAVING clause.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 10: Using Subqueries

Lab: Using Subqueries

Exercise 1: Writing Queries That Use Self-Contained Subqueries

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab10\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Write a SELECT Statement to Retrieve the Last Order Date

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab10\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard.)
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
SELECT MAX(orderdate) AS lastorderdate
FROM Sales.Orders;
```

7. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve All Orders Placed on the Last Order Date

1. In the query pane, type the following query after the task 2 description:

```
SELECT
orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE
orderdate = (SELECT MAX(orderdate) FROM Sales.Orders);
```

2. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Observe the T-SQL Statement Provided by the IT Department

1. Highlight the provided T-SQL statement under the task 3 description and click **Execute**.
2. Modify the query to filter customers whose contact name starts with the letter B. Your query should look like this:

```
SELECT
orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE
custid =
(
SELECT custid
FROM Sales.Customers
WHERE contactname LIKE N'B%'
);
```

3. Highlight the written query and click **Execute**.
4. Observe the error message:

> Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <= , >, >= or when the subquery is used as an expression.

Why did the query fail? It failed because the subquery returned more than one row. To fix this problem, you should replace the = operator with an IN operator.

5. Modify the query so that it uses the IN operator. Your query should look like this:

```
SELECT
orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE
custid IN
(
SELECT custid
FROM Sales.Customers
WHERE contactname LIKE N'B%'
);
```

6. Highlight the written query and click **Execute**.

▶ Task 5: Write A SELECT Statement to Analyze Each Order's Sales as a Percentage of the Total Sales Amount

1. In the query pane, type the following query after the task 4 description:

```
SELECT
o.orderid,
SUM(d.qty * d.unitprice) AS totalsalesamount,
SUM(d.qty * d.unitprice) /
(
SELECT SUM(d.qty * d.unitprice)
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.orderdate >= '20080501' AND orderdate < '20080601'
) * 100. AS salespctoftotal
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.orderdate >= '20080501' AND orderdate < '20080601'
GROUP BY o.orderid;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should be able to use self-contained subqueries in T-SQL statements.

Exercise 2: Writing Queries That Use Scalar and Multi-Result Subqueries

▶ Task 1: Write a SELECT Statement to Retrieve Specific Products

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
productid, productname
FROM Production.Products
WHERE
productid IN
(
SELECT productid
FROM Sales.OrderDetails
WHERE qty > 100
);
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Write a SELECT Statement to Retrieve Those Customers Without Orders

1. In the query pane, type the following query after the task 2 description:

```
SELECT
custid, contactname
FROM Sales.Customers
WHERE custid NOT IN
(
SELECT custid
FROM Sales.Orders
);
```

2. Highlight the written query and click **Execute**.
3. Observe the result. Notice there are two customers without an order.

▶ Task 3: Add a Row and Rerun the Query That Retrieves Those Customers Without Orders

1. Highlight the provided T-SQL statement under the task 3 description and click **Execute**. This code inserts an additional row that has a NULL in the custid column of the Sales.Orders table.
2. Highlight the query in task 2. On the toolbar, click **Edit** and then **Copy**.
3. In the query window, click the line after the provided T-SQL statement. On the toolbar, click **Edit** and then **Paste**.
4. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

5. Notice that you have an empty result despite getting two rows when you first ran the query in task 2. Why did you get an empty result this time? There is an issue with the NULL in the new row you added because the custid column is the only one that is part of the subquery. The IN operator supports three-valued logic (TRUE, FALSE, UNKNOWN). Before you apply the NOT operator, the logical meaning of UNKNOWN is that you can't tell for sure whether the customer ID appears in the set, because the NULL could represent that customer ID as well as anything else. As a more tangible example, consider the expression 22 NOT IN (1, 2, NULL). If you evaluate each individual expression in the parentheses to its truth value, you will get NOT (FALSE OR FALSE OR UNKNOWN), which translates to NOT UNKNOWN, which evaluates to UNKNOWN. The tricky part is that negating UNKNOWN with the NOT operator still yields UNKNOWN, and UNKNOWN is filtered out in a query filter. In short, when you use the NOT IN predicate against a subquery that returns at least one NULL, the outer query always returns an empty set.
6. To solve this problem, modify the T-SQL statement so that the subquery does not return NULLs. Your query should look like this:

```
SELECT
   custid, contactname
FROM Sales.Customers
WHERE custid NOT IN
   (
          SELECT custid
          FROM Sales.Orders
          WHERE custid IS NOT NULL
   );
```

7. Highlight the modified query and click **Execute**.

Results: After this exercise, you should know how to use multi-result subqueries in T-SQL statements.

Exercise 3: Writing Queries That Use Correlated Subqueries and an EXISTS Predicate

▶ Task 1: Write a SELECT Statement to Retrieve the Last Order Date for Each Customer

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
c.custid, c.contactname,
(
SELECT MAX(o.orderdate)
FROM Sales.Orders AS o
WHERE o.custid = c.custid
) AS lastorderdate
FROM Sales.Customers AS c;
```

4. Highlight the written query and click **Execute**.

MCT USE ONLY STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Write a SELECT Statement That Uses the EXISTS Predicate to Retrieve Those Customers Without Orders

1. In the query pane, type the following query after the task 2 description:

```
SELECT c.custid, c.contactname
FROM Sales.Customers AS c
WHERE
NOT EXISTS (SELECT * FROM Sales.Orders AS o WHERE o.custid = c.custid);
```

2. Highlight the written query and click **Execute**.

3. Notice that you achieved the same result as the modified query in exercise 2, task 3, but without a filter to exclude NULLs. Why didn't you need to explicitly filter out NULLs? The EXISTS predicate uses two-valued logic (TRUE, FALSE) and checks only if the rows specified in the correlated subquery exists. Another benefit of using the EXISTS predicate is better performance. The SQL Server engine knows it is enough to determine whether the subquery returns at least one row or none, so it doesn't need to process all qualifying rows.

▶ Task 3: Write a SELECT Statement to Retrieve Customers Who Bought Expensive Products

1. In the query pane, type the following query after the task 3 description:

```
SELECT c.custid, c.contactname
FROM Sales.Customers AS c
WHERE
EXISTS (
SELECT *
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.custid = c.custid
AND d.unitprice > 100.
AND o.orderdate >= '20080401'
);
```

2. Highlight the written query and click **Execute**.

▶ Task 4: Write a SELECT Statement to Display the Total Sales Amount and the Running Total Sales Amount for Each Order Year

1. In the query pane, type the following query after the task 4 description:

```
SELECT
YEAR(o.orderdate) as orderyear,
SUM(d.qty * d.unitprice) AS totalsales,
(
SELECT SUM(d2.qty * d2.unitprice)
FROM Sales.Orders AS o2
INNER JOIN Sales.OrderDetails AS d2 ON d2.orderid = o2.orderid
WHERE YEAR(o2.orderdate) <= YEAR(o.orderdate)
) AS runsales
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY YEAR(o.orderdate)
ORDER BY orderyear;
```

2. Highlight the written query and click **Execute**.

▶ Task 5: Clean the Sales.Customers Table

1. Highlight the provided T-SQL statement and click **Execute**.

Results: After this exercise, you should have an understanding of how to use a correlated subquery in T-SQL statements.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 11: Using Table Expressions

Lab: Using Table Expressions

Exercise 1: Writing Queries That Use Views

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab11\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Write a SELECT Statement to Retrieve All Products for a Specific Category

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab11\Starter\Project\Project.ssmssln**.
4. In Solution Explorer expand the **Queries** node, double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard).
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
SELECT
productid, productname, supplierid, unitprice, discontinued
FROM Production.Products
WHERE categoryid = 1;
```

7. Highlight the written query and click **Execute**.
8. Modify the query to include the provided CREATE VIEW statement. The query should look like this:

```
CREATE VIEW Production.ProductsBeverages AS
SELECT
productid, productname, supplierid, unitprice, discontinued
FROM Production.Products
WHERE categoryid = 1;
```

9. Highlight the modified query and click **Execute**.

▶ Task 3: Write a SELECT Statement Against the Created View

1. In the query pane, type the following query after the task 2 description:

```
SELECT
productid, productname
FROM Production.ProductsBeverages
WHERE supplierid = 1;
```

2. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Try to Use an ORDER BY Clause in the Created View

1. Highlight the provided T-SQL statement under the task 3 description and click **Execute**.
2. Observe the error message:

   Results: The ORDER BY clause is invalid in views, inline functions, derived tables, subqueries, and common table expressions, unless TOP, OFFSET or FOR XML is also specified.

   Why did the query fail? It failed because the view is supposed to represent a relation, and a relation has no order. You can only use the ORDER BY clause in the view if you specify the TOP, OFFSET, or FOR XML option. The reason you can use ORDER BY in special cases is that it serves a meaning other than presentation ordering to these special cases.
3. Modify the previous T-SQL statement by including the TOP (100) PERCENT option. The query should look like this:

```
ALTER VIEW Production.ProductsBeverages AS
SELECT TOP(100) PERCENT
productid, productname, supplierid, unitprice, discontinued
FROM Production.Products
WHERE categoryid = 1
ORDER BY productname;
```

4. Highlight the written query and click **Execute**.
5. Observe the result. If you now write a query against the Production.ProductsBeverages view, is it guaranteed that the retrieved rows will be sorted by productname? If you do not specify the ORDER BY clause in the T-SQL statement against the view, there is no guarantee that the retrieved rows will be sorted. It is important to remember that any order of the rows in the output is considered valid, and no specific order is guaranteed. Therefore, when querying a table expression, you should not assume any order unless you specify an ORDER BY clause in the outer query.

▶ Task 5: Add a Calculated Column to the View

1. Highlight the provided T-SQL statement under the task 4 description and click **Execute**.
2. Observe the error message:

   Results: Create View or Function failed because no column name was specified for column 6.
   Why did the query fail? It failed because each column must have a unique name. In the provided T-SQL statement, the last column does not have a name.
3. Modify the T-SQL statement to include the column name pricetype. The query should look like this:

```
ALTER VIEW Production.ProductsBeverages AS
SELECT
    productid, productname, supplierid, unitprice, discontinued,
    CASE WHEN unitprice > 100. THEN N'high' ELSE N'normal' END AS pricetype
FROM Production.Products
WHERE categoryid = 1;
```

4. Highlight the written query and click **Execute**.

▶ Task 6: Remove the Production.ProductsBeverages View

1. Highlight the provided T-SQL statement under the task 5 description and click **Execute**.

Results: After this exercise, you should know how to use a view in T-SQL statements.

Exercise 2: Writing Queries That Use Derived Tables

▶ Task 1: Write a SELECT Statement Against a Derived Table

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
p.productid, p.productname
FROM
(
SELECT
productid, productname, supplierid, unitprice, discontinued,
CASE WHEN unitprice > 100. THEN N'high' ELSE N'normal' END AS pricetype
FROM Production.Products
WHERE categoryid = 1
) AS p
WHERE p.pricetype = N'high';
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Write a SELECT Statement to Calculate the Total and Average Sales Amount

1. In the query pane, type the following query after the task 2 description:

```
SELECT
c.custid,
SUM(c.totalsalesamountperorder) AS totalsalesamount,
AVG(c.totalsalesamountperorder) AS avgsalesamount
FROM
(
SELECT
o.custid, o.orderid, SUM(d.unitprice * d.qty) AS totalsalesamountperorder
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails d ON d.orderid = o.orderid
GROUP BY o.custid, o.orderid
) AS c
GROUP BY c.custid;
```

2. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Write a SELECT Statement to Retrieve the Sales Growth Percentage

1. In the query pane, type the following query after the task 3 description:

```
SELECT
cy.orderyear,
cy.totalsalesamount AS curtotalsales,
py.totalsalesamount AS prevtotalsales,
(cy.totalsalesamount - py.totalsalesamount) / py.totalsalesamount * 100. AS
percentgrowth
FROM
(
SELECT
YEAR(orderdate) AS orderyear, SUM(val) AS totalsalesamount
FROM Sales.OrderValues
GROUP BY YEAR(orderdate)
) AS cy
LEFT OUTER JOIN
(
SELECT
YEAR(orderdate) AS orderyear, SUM(val) AS totalsalesamount
FROM Sales.OrderValues
GROUP BY YEAR(orderdate)
) AS py ON cy.orderyear = py.orderyear + 1
ORDER BY cy.orderyear;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should be able to use derived tables in T-SQL statements.

Exercise 3: Writing Queries That Use CTEs

▶ Task 1: Write a SELECT Statement that Uses a CTE

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
WITH ProductsBeverages AS
(
SELECT
productid, productname, supplierid, unitprice, discontinued,
CASE WHEN unitprice > 100. THEN N'high' ELSE N'normal' END AS pricetype
FROM Production.Products
WHERE categoryid = 1
)
SELECT
productid, productname
FROM ProductsBeverages
WHERE pricetype = N'high';
```

4. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer

1. In the query pane, type the following query after the task 2 description:

```
WITH c2008 (custid, salesamt2008) AS
(
SELECT
custid, SUM(val)
FROM Sales.OrderValues
WHERE YEAR(orderdate) = 2008
GROUP BY custid
)
SELECT
c.custid, c.contactname, c2008.salesamt2008
FROM Sales.Customers AS c
LEFT OUTER JOIN c2008 ON c.custid = c2008.custid;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Compare the Total Sales Amount for Each Customer Over the Previous Year

1. In the query pane, type the following query after the task 3 description:

```
WITH c2008 (custid, salesamt2008) AS
(
SELECT
custid, SUM(val)
FROM Sales.OrderValues
WHERE YEAR(orderdate) = 2008
GROUP BY custid
),
c2007 (custid, salesamt2007) AS
(
SELECT
custid, SUM(val)
FROM Sales.OrderValues
WHERE YEAR(orderdate) = 2007
GROUP BY custid
)
SELECT
c.custid, c.contactname,
c2008.salesamt2008,
c2007.salesamt2007,
COALESCE((c2008.salesamt2008 - c2007.salesamt2007) / c2007.salesamt2007 * 100., 0) AS
percentgrowth
FROM Sales.Customers AS c
LEFT OUTER JOIN c2008 ON c.custid = c2008.custid
LEFT OUTER JOIN c2007 ON c.custid = c2007.custid
ORDER BY percentgrowth DESC;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should have an understanding of how to use a CTE in a T-SQL statement.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 4: Writing Queries That Use Inline TVFs

▶ Task 1: Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer

1. In Solution Explorer, double-click the query **81 - Lab Exercise 4.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
custid, SUM(val) AS totalsalesamount
FROM Sales.OrderValues
WHERE YEAR(orderdate) = 2007
GROUP BY custid;
```

4. Highlight the written query and click **Execute**.
5. Create an inline TVF using the provided code. Add the previous query, putting it after the function's RETURN clause. In the query, replace the order date of 2007 with the function's input parameter @orderyear. The resulting T-SQL statement should look like this:

```
CREATE FUNCTION dbo.fnGetSalesByCustomer
(@orderyear AS INT) RETURNS TABLE
AS
RETURN
SELECT
custid, SUM(val) AS totalsalesamount
FROM Sales.OrderValues
WHERE YEAR(orderdate) = @orderyear
GROUP BY custid;
```

   This T-SQL statement will create an inline TVF named dbo.fnGetSalesByCustomer.

6. Highlight the written T-SQL statement and click **Execute**.

▶ Task 2: Write a SELECT Statement Against the Inline TVF

1. In the query pane, type the following query after the task 2 description:

```
SELECT
custid, totalsalesamount
FROM dbo.fnGetSalesByCustomer(2007);
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve the Top Three Products Based on the Total Sales Value for a Specific Customer

1. In the query pane, type the following query after the task 3 description:

```
SELECT TOP(3)
d.productid,
MAX(p.productname) AS productname,
SUM(d.qty * d.unitprice) AS totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
INNER JOIN Production.Products AS p ON p.productid = d.productid
WHERE custid = 1
GROUP BY d.productid
ORDER BY totalsalesamount DESC;
```

2. Highlight the written query and click **Execute**.
3. Create an inline TVF using the provided code. Add the previous query, putting it after the function's RETURN clause. In the query, replace the constant custid value of 1 with the function's input parameter @custid. The resulting T-SQL statement should look like this:

```
CREATE FUNCTION dbo.fnGetTop3ProductsForCustomer
(@custid AS INT) RETURNS TABLE
AS
RETURN
SELECT TOP(3)
d.productid,
MAX(p.productname) AS productname,
SUM(d.qty * d.unitprice) AS totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
INNER JOIN Production.Products AS p ON p.productid = d.productid
WHERE custid = @custid
GROUP BY d.productid
ORDER BY totalsalesamount DESC;
```

4. To test the inline TVF, add the following query after the CREATE FUNCTION and GO statement:

```
SELECT
p.productid,
p.productname,
p.totalsalesamount
FROM dbo.fnGetTop3ProductsForCustomer(1) AS p;
```

5. Highlight the CREATE FUNCTION statement and the written query, and click **Execute**.

▶ Task 4: Using Inline TVFs, Write a SELECT Statement to Compare the Total Sales Amount for Each Customer Over the Previous Year

1. In the query pane, type the following query after the task 4 description:

```
SELECT
c.custid, c.contactname,
c2008.totalsalesamount AS salesamt2008,
c2007.totalsalesamount AS salesamt2007,
COALESCE((c2008.totalsalesamount - c2007.totalsalesamount) / c2007.totalsalesamount *
100., 0) AS percentgrowth
FROM Sales.Customers AS c
LEFT OUTER JOIN dbo.fnGetSalesByCustomer(2007) AS c2007 ON c.custid = c2007.custid
LEFT OUTER JOIN dbo.fnGetSalesByCustomer(2008) AS c2008 ON c.custid = c2008.custid;
```

2. Highlight the written query and click **Execute**.

▶ Task 5: Remove the Created Inline TVFs

1. Highlight the provided T-SQL statement under the task 5 description and click **Execute**.

Results: After this exercise, you should know how to use inline TVFs in T-SQL statements.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 12: Using Set Operators

Lab: Using Set Operators

Exercise 1: Writing Queries That Use UNION Set Operators and UNION ALL Multi-Set Operators

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab12\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish

▶ Task 2: Write a SELECT Statement to Retrieve Specific Products

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab12\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand **Queries** folder and double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard.)
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
SELECT
productid, productname
FROM Production.Products
WHERE categoryid = 4;
```

7. Highlight the written query and click **Execute**. Observe that the query retrieved 10 rows.

▶ Task 3: Write a SELECT Statement to Retrieve All Products with a Total Sales Amount of More than $50,000

1. In the query pane, type the following query after the task 2 description:

```
SELECT
d.productid, p.productname
FROM Sales.OrderDetails d
INNER JOIN Production.Products p ON p.productid = d.productid
GROUP BY d.productid, p.productname
HAVING SUM(d.qty * d.unitprice) > 50000;
```

2. Highlight the written query and click **Execute**. Observe that the query retrieved four rows.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Merge the Results from Task 1 and Task 2

1. In the query pane, type the following query after the task 3 description:

```
SELECT
productid, productname
FROM Production.Products
WHERE categoryid = 4
UNION
SELECT
d.productid, p.productname
FROM Sales.OrderDetails d
INNER JOIN Production.Products p ON p.productid = d.productid
GROUP BY d.productid, p.productname
HAVING SUM(d.qty * d.unitprice) > 50000;
```

2. Highlight the written query and click **Execute**.
3. Observe the result. What is the total number of rows in the result? If you compare this number with an aggregate value of the number of rows from tasks 1 and 2, is there any difference? The total number of rows retrieved by the query is 12. This is two rows less than the aggregate value of rows from the query in task 1 (10 rows) and task 2 (four rows).
4. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
5. In the query window, click the line after the written T-SQL statement. On the toolbar, click **Edit** and then **Paste**.
6. Modify the T-SQL statement by replacing the UNION operator with the UNION ALL operator. The query should look like this:

```
SELECT
productid, productname
FROM Production.Products
WHERE categoryid = 4
UNION ALL
SELECT
d.productid, p.productname
FROM Sales.OrderDetails d
INNER JOIN Production.Products p ON p.productid = d.productid
GROUP BY d.productid, p.productname
HAVING SUM(d.qty * d.unitprice) > 50000;
```

7. Highlight the modified query and click **Execute**.
8. Observe the result. What is the total number of rows in the result? What is the difference between the UNION and UNION ALL operators? The total number of rows retrieved by the query is 14. It is the same as the aggregate value of rows from the queries in tasks 1 and 2. This is because UNION ALL is a multi-set operator that returns all rows that appear in any of the inputs, without really comparing rows and without eliminating duplicates. The UNION set operator removes the duplicate rows and the result consists of only distinct rows.
9. So, when should you use either UNION ALL or UNION when unifying two inputs? If a potential exists for duplicates and you need to return them, use UNION ALL. If a potential exists for duplicates but you need to return distinct rows, use UNION. If no potential exists for duplicates when unifying the two inputs, UNION and UNION ALL are logically equivalent. However, in such a case, using UNION ALL is recommended because it removes the overhead of SQL Server checking for duplicates.

▶ Task 5: Write a SELECT Statement to Retrieve the Top 10 Customers by Sales Amount for January 2008 and February 2008

1. In the query pane, type the following query after the task 4 description:

```
SELECT
c1.custid, c1.contactname
FROM
(
SELECT TOP (10)
o.custid, c.contactname
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE o.orderdate >= '20080101' AND o.orderdate < '20080201'
GROUP BY o.custid, c.contactname
ORDER BY SUM(o.val) DESC
) AS c1
UNION
SELECT c2.custid, c2.contactname
FROM
(
SELECT TOP (10)
o.custid, c.contactname
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE o.orderdate >= '20080201' AND o.orderdate < '20080301'
GROUP BY o.custid, c.contactname
ORDER BY SUM(o.val) DESC
) AS c2;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should know how to use the UNION and UNION ALL set operators in T-SQL statements.

Exercise 2: Writing Queries That Use the CROSS APPLY and OUTER APPLY Operators

▶ Task 1: Write a SELECT Statement That Uses the CROSS APPLY Operator to Retrieve the Last Two Orders for Each Product

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
p.productid, p.productname, o.orderid
FROM Production.Products AS p
CROSS APPLY
(
SELECT TOP(2)
d.orderid
FROM Sales.OrderDetails AS d
WHERE d.productid = p.productid
ORDER BY d.orderid DESC
) o
ORDER BY p.productid;
```

4. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Write a SELECT Statement That Uses the CROSS APPLY Operator to Retrieve the Top Three Products, Based on Sales Revenue, for Each Customer

1. Highlight the provided T-SQL code after the task 2 description and click **Execute**.
2. In the query pane, type the following query after the provided T-SQL code:

```
SELECT
c.custid, c.contactname, p.productid, p.productname, p.totalsalesamount
FROM Sales.Customers AS c
CROSS APPLY dbo.fnGetTop3ProductsForCustomer (c.custid) AS p
ORDER BY c.custid;
```

   Tip: You can make the inline TVF (dbo.fnGetTop3ProductsForCustomer) more flexible by making the number of top rows to return an argument instead of fixing the number to three in the function's code.

3. Highlight the written query and click **Execute**. The query retrieved 265 rows.

▶ Task 3: Use the OUTER APPLY Operator

1. Highlight the previous query in task 2. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 3 description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the T-SQL statement by replacing the CROSS APPLY operator with the OUTER APPLY operator. The query should look like this:

```
SELECT
   c.custid, c.contactname, p.productid, p.productname, p.totalsalesamount
FROM Sales.Customers AS c
OUTER APPLY dbo.fnGetTop3ProductsForCustomer (c.custid) AS p
ORDER BY c.custid;
```

4. Highlight the modified query and click **Execute**.
5. Notice that the query retrieved 267 rows, which is two more rows than the previous query. If you observe the result, you will notice two rows with NULL in the columns from the inline TVF.

▶ Task 4: Analyze the OUTER APPLY Operator

1. Highlight the previous query in task 3. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 4 description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the T-SQL statement to search for a null productid. The query should look like this:

```
SELECT
c.custid, c.contactname, p.productid, p.productname, p.totalsalesamount
FROM Sales.Customers AS c
OUTER APPLY dbo.fnGetTop3ProductsForCustomer (c.custid) AS p
WHERE p.productid IS NULL;
```

4. Highlight the modified query and click **Execute**.
5. Notice that the query now retrieves the two rows that do not occur in the CROSS APPLY query in Task 2.

▶ Task 5: Remove the Created Inline TVF

1. Highlight the provided T-SQL statement after the task 5 description and click **Execute**.

Results: After this exercise, you should be able to use the CROSS APPLY and OUTER APPLY operators in your T-SQL statements.

Exercise 3: Writing Queries That Use the EXCEPT and INTERSECT Operators

▶ Task 1: Write a SELECT Statement to Return All Customers Who Bought More than 20 Distinct Products

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING COUNT(DISTINCT d.productid) > 20;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Write a SELECT Statement to Retrieve All Customers from the USA, Except Those Who Bought More than 20 Distinct Products

1. In the query pane, type the following query after the task 2 description:

```
SELECT
custid
FROM Sales.Customers
WHERE country = 'USA'
EXCEPT
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING COUNT(DISTINCT d.productid) > 20;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve Customers Who Spent More than $10,000

1. In the query pane, type the following query after the task 3 description:

```
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING SUM(d.qty * d.unitprice) > 10000;
```

2. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Write a SELECT Statement That Uses the EXCEPT and INTERSECT Operators

1. Highlight the query from task 2. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 4 description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the first SELECT statement so that it selects all customers – not just those from the USA – and include the INTERSECT operator, adding the query from task 3. The query should look like this:

```
SELECT
c.custid
FROM Sales.Customers AS c
EXCEPT
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING COUNT(DISTINCT d.productid) > 20
INTERSECT
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING SUM(d.qty * d.unitprice) > 10000;
```

4. Highlight the modified query and click **Execute**.
5. Observe that the total number of rows is 59. In business terms, can you explain in which customers are part of the result? Because the INTERSECT operator is evaluated before the EXCEPT operator, the result consists of all customers, except those who bought more than 20 different products and spent more than $10,000.

▶ Task 5: Change the Operator Precedence

1. Highlight the previous query in task 4. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 5 description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the T-SQL statement by adding a set of parentheses around the first two SELECT statements. The query should look like this:

```
(
SELECT
c.custid
FROM Sales.Customers AS c
EXCEPT
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING COUNT(DISTINCT d.productid) > 20
)
INTERSECT
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING SUM(d.qty * d.unitprice) > 10000;
```

4. Highlight the provided T-SQL statement and click **Execute**.
5. Observe that the total number of rows is nine. Is that different to the result of the query in task 4? Yes, because when you added the parentheses, the SQL Server engine first evaluated the EXCEPT operation, and then the INTERSECT operation. In business terms, this query retrieved all customers who did not buy more than 20 distinct products and who spent more than $10,000.
6. What is the precedence among the set operators? SQL defines the following precedence among the set operations: INTERSECT precedes UNION and EXCEPT, while UNION and EXCEPT are considered equal. In a query that contains multiple set operations, INTERSECT operations are evaluated first, and then operations with the same precedence are evaluated, based on appearance order. Remember that set operations in parentheses precede all.

Results: After this exercise, you should have an understanding of how to use the EXCEPT and INTERSECT operators in T-SQL statements.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 13: Using Window Ranking, Offset, and Aggregate Functions

Lab: Using Window Ranking, Offset, and Aggregate Functions

Exercise 1: Writing Queries That Use Ranking Functions

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab13\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish

▶ Task 2: Write a SELECT Statement That Uses the ROW_NUMBER Function to Create a Calculated Column

1. Start SQL Server Management Studio and connect to the MIA-SQL database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab13\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard).
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
SELECT
orderid,
orderdate,
val,
ROW_NUMBER() OVER (ORDER BY orderdate) AS rowno
FROM Sales.OrderValues;
```

7. Highlight the written query and click **Execute**.

▶ Task 3: Add an Additional Column Using the RANK Function

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 2 description. On the toolbar, click **Edit** and then **Paste**.

MCT USE ONLY. STUDENT USE PROHIBITED

3. Modify the T-SQL statement by adding an additional calculated column. The query should look like this:

```
SELECT
orderid,
orderdate,
val,
ROW_NUMBER() OVER (ORDER BY orderdate) AS rowno,
RANK() OVER (ORDER BY orderdate) AS rankno
FROM Sales.OrderValues;
```

4. Highlight the written query and click **Execute**.
5. Observe the results. What is the difference between the RANK and ROW_NUMBER functions? The ROW_NUMBER function provides unique sequential integer values within the partition. The RANK function assigns the same ranking value to rows with the same values in the specified sort columns when the ORDER BY list is not unique. Also, the RANK function skips the next number if there is a tie in the ranking value.

▶ Task 4: Write A SELECT Statement to Calculate a Rank, Partitioned by Customer and Ordered by the Order Value

1. In the query pane, type the following query after the task 3 description:

```
SELECT
orderid,
orderdate,
custid,
val,
RANK() OVER (PARTITION BY custid ORDER BY val DESC) AS orderrankno FROM
Sales.OrderValues;
```

2. Highlight the written query and click **Execute**.

▶ Task 5: Write a SELECT Statement to Rank Orders, Partitioned by Customer and Order Year, and Ordered by the Order Value

1. In the query pane, type the following query after the task 4 description:

```
SELECT
custid,
val,
YEAR(orderdate) as orderyear,
RANK() OVER (PARTITION BY custid, YEAR(orderdate) ORDER BY val DESC) AS orderrankno
FROM Sales.OrderValues;
```

2. Highlight the written query and click **Execute**.

▶ Task 6: Filter Only Orders with the Top Two Ranks

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 5 description. On the toolbar, click **Edit** and then **Paste**.

MCT USE ONLY. STUDENT USE PROHIBITED

3. Modify the T-SQL statement to look like this:

```
SELECT
s.custid,
s.orderyear,
s.orderrankno,
s.val
FROM
(
SELECT
custid,
val,
YEAR(orderdate) as orderyear,
RANK() OVER (PARTITION BY custid, YEAR(orderdate) ORDER BY val DESC) AS orderrankno
FROM Sales.OrderValues
) AS s
WHERE s.orderrankno <= 2;
```

4. Highlight the written query and click **Execute**.

Results: After this exercise, you should know how to use ranking functions in T-SQL statements.

Exercise 2: Writing Queries That Use Offset Functions

▶ Task 1: Write a SELECT Statement to Retrieve the Next Row Using a Common Table Expression (CTE)

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
WITH OrderRows AS
(
SELECT
orderid,
orderdate,
ROW_NUMBER() OVER (ORDER BY orderdate, orderid) AS rowno,
val
FROM Sales.OrderValues
)
SELECT
o.orderid,
o.orderdate,
o.val,
o2.val as prevval,
o.val - o2.val as diffprev
FROM OrderRows AS o
LEFT OUTER JOIN OrderRows AS o2 ON o.rowno = o2.rowno + 1;
```

4. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Add a Column to Display the Running Sales Total

1. In the query pane, type the following query after the Task 2 description:

```
SELECT
orderid,
orderdate,
val,
LAG(val) OVER (ORDER BY orderdate, orderid) AS prevval,
val - LAG(val) OVER (ORDER BY orderdate, orderid) AS diffprev
FROM Sales.OrderValues;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Analyze the Sales Information for the Year 2007

1. In the query pane, type the following query after the task 3 description:

```
WITH SalesMonth2007 AS
(
SELECT
MONTH(orderdate) AS monthno,
SUM(val) AS val
FROM Sales.OrderValues
WHERE orderdate >= '20070101' AND orderdate < '20080101'
GROUP BY MONTH(orderdate)
)
SELECT
monthno,
val,
(LAG(val, 1, 0) OVER (ORDER BY monthno) + LAG(val, 2, 0) OVER (ORDER BY monthno) +
LAG(val, 3, 0) OVER (ORDER BY monthno)) / 3 AS avglast3months,
val - FIRST_VALUE(val) OVER (ORDER BY monthno ROWS UNBOUNDED PRECEDING) AS
diffjanuary,
LEAD(val) OVER (ORDER BY monthno) AS nextval
FROM SalesMonth2007;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should be able to use the offset functions in your T-SQL statements.

Exercise 3: Writing Queries That Use Window Aggregate Functions

▶ Task 1: Write a SELECT Statement to Display the Contribution of Each Customer's Order Compared to That Customer's Total Purchase

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
custid,
orderid,
orderdate,
val,
100. * val / SUM(val) OVER (PARTITION BY custid) AS percoftotalcust
FROM Sales.OrderValues
ORDER BY custid, percoftotalcust DESC;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Add a Column to Display the Running Sales Total

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 2 description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the T-SQL statement by adding an additional calculated column. The query should look like this:

```
SELECT
custid,
orderid,
orderdate,
val,
100. * val / SUM(val) OVER (PARTITION BY custid) AS percoftotalcust,
SUM(val) OVER (PARTITION BY custid
ORDER BY orderdate, orderid
ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW) AS runval
FROM Sales.OrderValues;
```

4. Highlight the written query and click **Execute**.

▶ Task 3: Analyze the Year-to-Date Sales Amount and Average Sales Amount for the Last Three Months

1. In the query pane, type the following query after the task 3 description:

```
WITH SalesMonth2007 AS
(
SELECT
MONTH(orderdate) AS monthno,
SUM(val) AS val
FROM Sales.OrderValues
WHERE orderdate >= '20070101' AND orderdate < '20080101'
GROUP BY MONTH(orderdate)
)
SELECT
monthno,
val,
AVG(val) OVER (ORDER BY monthno ROWS BETWEEN 3 PRECEDING AND CURRENT ROW) AS
avglast3months,
SUM(val) OVER (ORDER BY monthno ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
ytdval
FROM SalesMonth2007;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should have a basic understanding of how to use window aggregate functions in T-SQL statements.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 14: Pivoting and Grouping Sets

Lab: Pivoting and Grouping Sets

Exercise 1: Writing Queries That Use the PIVOT Operator

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab14\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Write a SELECT Statement to Retrieve the Number of Customers for a Specific Customer Group

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab14\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard.)
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar.
6. Highlight the following provided T-SQL code:

```
CREATE VIEW Sales.CustGroups AS
SELECT
custid,
CHOOSE(custid % 3 + 1, N'A', N'B', N'C') AS custgroup,
Country
FROM Sales.Customers;
```

7. Click **Execute**. This code creates a view named Sales.CustGroups.
8. In the query pane, type the following query after the provided T-SQL code:

```
SELECT
custid,
custgroup,
country
FROM Sales.CustGroups;
```

9. Highlight the written query and click **Execute**.
10. Modify the written T-SQL code by applying the PIVOT operator. The query should look like this:

```
SELECT
country,
p.A,
p.B,
p.C
FROM Sales.CustGroups
PIVOT (COUNT(custid) FOR custgroup IN (A, B, C)) AS p;
```

11. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Specify the Grouping Element for the PIVOT Operator

1. Highlight the following provided T-SQL code after the Task 2 description:

```
ALTER VIEW Sales.CustGroups AS
SELECT
custid,
CHOOSE(custid % 3 + 1, N'A', N'B', N'C') AS custgroup,
country,
city,
contactname
FROM Sales.Customers;
```

2. Click **Execute**. This code modifies the view by adding two additional columns.
3. Highlight the last query in task 1. On the toolbar, click **Edit** and then **Copy**.
4. In the query window, click the line after the provided T-SQL code. On the toolbar, click **Edit** and then **Paste**. The query should look like this:

```
SELECT
country,
p.A,
p.B,
p.C
FROM Sales.CustGroups
PIVOT (COUNT(custid) FOR custgroup IN (A, B, C)) AS p;
```

5. Highlight the copied query and click **Execute**.
6. Observe the result. Is this result the same as that from the query in task 1? The result is not the same. More rows were returned after you modified the view.
7. Modify the copied T-SQL statement to include additional columns from the view. The query should look like this:

```
SELECT
country,
city,
contactname,
p.A,
p.B,
p.C
FROM Sales.CustGroups
PIVOT (COUNT(custid) FOR custgroup IN (A, B, C)) AS p;
```

8. Highlight the written query and click **Execute**.
9. Notice that you received the same result as the previous query. Why did you get the same number of rows? The PIVOT operator assumes that all the columns except the aggregation and spreading elements are part of the grouping columns.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Use a Common Table Expression (CTE) to Specify the Grouping Element for the PIVOT Operator

1. In the query pane, type the following query after the task 3 description:

```
WITH PivotCustGroups AS
(
SELECT
custid,
country,
custgroup
FROM Sales.CustGroups
)
SELECT
country,
p.A,
p.B,
p.C
FROM PivotCustGroups
PIVOT (COUNT(custid) FOR custgroup IN (A, B, C)) AS p;
```

2. Highlight the written query and click **Execute**.
3. Observe the result. Is it the same as the result of the last query in task 1? Can you explain why? The result is the same. In this task, the CTE has provided three possible columns to the PIVOT operator. In task 1, the view also provided three columns to the PIVOT operator.
4. Why do you think it is beneficial to use a CTE when using the PIVOT operator? When using the PIVOT operator, you cannot directly specify the grouping element since SQL Server automatically assumes that all columns should be used as grouping elements, with the exception of the spreading and aggregation elements. With a CTE, you can specify the exact columns and therefore control which columns to use for the grouping.

▶ Task 5: Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer and Product Category

1. In the query pane, type the following query after the task 4 description:

```
WITH SalesByCategory AS
(
SELECT
o.custid,
d.qty * d.unitprice AS salesvalue,
c.categoryname
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON o.orderid = d.orderid
INNER JOIN Production.Products AS p ON p.productid = d.productid
INNER JOIN Production.Categories AS c ON c.categoryid = p.categoryid
WHERE o.orderdate >= '20080101' AND o.orderdate < '20090101'
)
SELECT
custid,
p.Beverages,
p.Condiments,
p.Confections,
p.[Dairy Products],
p.[Grains/Cereals],
p.[Meat/Poultry],
p.Produce,
p.Seafood
FROM SalesByCategory
PIVOT (SUM(salesvalue) FOR categoryname
IN (Beverages, Condiments, Confections, [Dairy Products], [Grains/Cereals],
[Meat/Poultry], Produce, Seafood)) AS p;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should be able to use the PIVOT operator in T-SQL statements.

Exercise 2: Writing Queries That Use the UNPIVOT Operator

▶ Task 1: Create and Query the Sales.PivotCustGroups View

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. Highlight the following provided T-SQL code:

```
CREATE VIEW Sales.PivotCustGroups AS
WITH PivotCustGroups AS
(
SELECT
custid,
country,
custgroup
FROM Sales.CustGroups
)
SELECT
country,
p.A,
p.B,
p.C
FROM PivotCustGroups
PIVOT (COUNT(custid) FOR custgroup IN (A, B, C)) AS p;
```

4. Click **Execute**. This code creates a view named Sales.PivotCustGroups.
5. In the query pane, type the following query after the provided T-SQL code:

```
SELECT
country, A, B, C
FROM Sales.PivotCustGroups;
```

6. Highlight the written query and click **Execute**.

▶ Task 2: Write a SELECT Statement to Retrieve a Row for Each Country and Customer Group

1. In the query pane, type the following query after the Task 2 descriptions:

```
SELECT
custgroup,
country,
numberofcustomers
FROM Sales.PivotCustGroups
UNPIVOT (numberofcustomers FOR custgroup IN (A, B, C)) AS p;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Remove the Created Views

1. Highlight the provided T-SQL statement after Task 3 description and click **Execute**.

Results: After this exercise, you should know how to use the UNPIVOT operator in your T-SQL statements.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Writing Queries That Use the GROUPING SETS, CUBE, and ROLLUP Subclauses

▶ Task 1: Write a SELECT Statement That Uses the GROUPING SETS Subclause to Return the Number of Customers for Different Grouping Sets

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
country,
city,
COUNT(custid) AS noofcustomers
FROM Sales.Customers
GROUP BY
GROUPING SETS
(
(country, city),
(country),
(city),
()
);
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Write a SELECT Statement That Uses the CUBE Subclause to Retrieve Grouping Sets Based on Yearly, Monthly, and Daily Sales Values

1. In the query pane, type the following query after the task 2 description:

```
SELECT
YEAR(orderdate) AS orderyear,
MONTH(orderdate) AS ordermonth,
DAY(orderdate) AS orderday,
SUM(val) AS salesvalue
FROM Sales.OrderValues
GROUP BY
CUBE (YEAR(orderdate), MONTH(orderdate), DAY(orderdate));
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Write the Same SELECT Statement Using the ROLLUP Subclause

1. In the query pane, type the following query after the task 3 description:

```
SELECT
YEAR(orderdate) AS orderyear,
MONTH(orderdate) AS ordermonth,
DAY(orderdate) AS orderday,
SUM(val) AS salesvalue
FROM Sales.OrderValues
GROUP BY
ROLLUP (YEAR(orderdate), MONTH(orderdate), DAY(orderdate));
```

2. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

3. Observe the result. What is the difference between the ROLLUP and CUBE subclauses of the GROUP BY clause? Like the CUBE subclause, the ROLLUP subclause provides an abbreviated way to define multiple grouping sets. However, unlike CUBE, ROLLUP doesn't produce all possible grouping sets that can be defined based on the input members—it produces a subset of those. ROLLUP assumes a hierarchy among the input members and produces all grouping sets that make sense, considering the hierarchy. In other words, while CUBE(a, b, c) produces all eight possible grouping sets out of the three input members, ROLLUP(a, b, c) produces only four grouping sets, assuming the hierarchy a>b>c. ROLLUP(a, b, c) is the equivalent of specifying GROUPING SETS((a, b, c), (a, b), (a), ()).

   Which is the more appropriate subclause to use in this example? Since year, month, and day form a hierarchy, the ROLLUP clause is more suitable. There is probably not much interest in showing aggregates for a month irrespective of year, but the other way around is interesting.

▶ Task 4: Analyze the Total Sales Value by Year and Month

1. In the query pane, type the following query after the task 4 description:

```
SELECT
GROUPING_ID(YEAR(orderdate), MONTH(orderdate)) as groupid,
YEAR(orderdate) AS orderyear,
MONTH(orderdate) AS ordermonth,
SUM(val) AS salesvalue
FROM Sales.OrderValues
GROUP BY
ROLLUP (YEAR(orderdate), MONTH(orderdate))
ORDER BY groupid, orderyear, ordermonth;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should have an understanding of how to use the GROUPING SETS, CUBE, and ROLLUP subclauses in T-SQL statements.

MCT USE ONLY STUDENT USE PROHIBITED

Module 15: Executing Stored Procedures

Lab: Executing Stored Procedures

Exercise 1: Using the EXECUTE Statement to Invoke Stored Procedures

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab15\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Create and Execute a Stored Procedure

1. Start SQL Server Management Studio and connect to the MIA-SQL database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab15\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand the **Queries** node then double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard.)
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. Highlight the following T-SQL code under the task 1 description:

```
CREATE PROCEDURE Sales.GetTopCustomers AS
SELECT TOP(10)
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC;
```

7. Click **Execute**. You have created a stored procedure named Sales.GetTopCustomers.
8. In the query pane, type the following T-SQL code after the previous T-SQL code:

```
EXECUTE Sales.GetTopCustomers;
```

9. Highlight the written T-SQL code and click **Execute**. You have executed the stored procedure.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Modify the Stored Procedure and Execute It

1. Highlight the following T-SQL code after the task 2 description:

```
ALTER PROCEDURE Sales.GetTopCustomers AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

2. Click **Execute**. You have modified the Sales.GetTopCustomers stored procedure.
3. In the query pane, type the following T-SQL code after the previous T-SQL code:

```
EXECUTE Sales.GetTopCustomers;
```

4. Highlight the written T-SQL code and click **Execute**. You have executed the modified stored procedure.
5. Compare both the code and the result of the two versions of the stored procedure. What is the difference between them? In the modified version, the TOP option has been replaced with the OFFSET-FETCH option. Despite this change, the result is the same.

   If some applications had been using the stored procedure in task 1, would they still work properly after the change you applied in task 2? Yes, since the result from the stored procedure is still the same. This demonstrates the huge benefit of using stored procedures as an additional layer between the database and the application/middle tier. Even if you change the underlying T-SQL code, the application would work properly without any changes. There are also other benefits of using stored procedures in terms of performance (for example, caching and reuse of plans) and security (for example, preventing SQL injections).

Results: After this exercise, you should be able to invoke a stored procedure using the EXECUTE statement.

MCT USE ONLY STUDENT USE PROHIBITED

Exercise 2: Passing Parameters to Stored Procedures

▶ Task 1: Execute a Stored Procedure with a Parameter for Order Year

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. Highlight the following T-SQL code under the task 1 description:

```
ALTER PROCEDURE Sales.GetTopCustomers
@orderyear int
AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE YEAR(o.orderdate) = @orderyear
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

4. Click **Execute**. You have modified the Sales.GetTopCustomers stored procedure to accept the parameter @orderyear. Notice that the modified stored procedure uses a predicate in the WHERE clause that isn't a search argument. This predicate was used to keep things simple. The best practice is to avoid such filtering because it does not allow efficient use of indexing. A better approach would be to use the DATETIMEFROMPARTS function to provide a search argument for orderdate:

```
WHERE o.orderdate >= DATETIMEFROMPARTS(@orderyear, 1, 1, 0, 0, 0, 0)
AND o.orderdate < DATETIMEFROMPARTS(@orderyear + 1, 1, 1, 0, 0, 0, 0)
```

5. In the query pane, type the following T-SQL code after the previous T-SQL code:

```
EXECUTE Sales.GetTopCustomers @orderyear = 2007;
```

   Notice that you are passing the parameter by name as this is considered the best practice. There is also support for passing parameters by position. For example, the following EXECUTE statement would retrieve the same result as the T-SQL code you just typed:

```
EXECUTE Sales.GetTopCustomers 2007;
```

6. Highlight the written T-SQL code and click **Execute**.
7. After the previous T-SQL code, type the following T-SQL code to execute the stored procedure for the order year 2008:

```
EXECUTE Sales.GetTopCustomers @orderyear = 2008;
```

8. Highlight the written T-SQL code and click **Execute**.
9. After the previous T-SQL code, type the following T-SQL code to execute the stored procedure without specifying a parameter:

```
EXECUTE Sales.GetTopCustomers;
```

10. Highlight the written T-SQL code and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

11. Observe the error message:

    Procedure or function 'GetTopCustomers' expects parameter '@orderyear', which was not supplied.

    This error message is telling you that the @orderyear parameter was not supplied.

12. Suppose that an application named MyCustomers is using the exercise 1 version of the stored procedure. Would the modification made to the stored procedure in this exercise impact the usability of the GetCustomerInfo application? Yes. The exercise 1 version of the stored procedure did not need a parameter, whereas the version in this exercise does not work without a parameter. To avoid problems, you can add a default parameter to the stored procedure. That way, the MyCustomers application does not have to be changed to support the @orderyear parameter.

▶ Task 2: Modify the Stored Procedure to have a Default Value for the Parameter

1. Highlight the following T-SQL code under the task 2 description:

```
ALTER PROCEDURE Sales.GetTopCustomers
@orderyear int = NULL
AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE YEAR(o.orderdate) = @orderyear OR @orderyear IS NULL
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

2. Click **Execute**. You have modified the Sales.GetTopCustomers stored procedure to have a default value (NULL) for the @orderyear parameter. You have also included an additional logical expression to the WHERE clause.

3. In the query pane, type the following T-SQL code after the previous one:

```
EXECUTE Sales.GetTopCustomers;
```

    This code tests the modified stored procedure by executing it without specifying a parameter.

4. Highlight the written query and click **Execute**.

5. Observe the result. How do the changes to the stored procedure in task 2 influence the MyCustomers application and the design of future applications? The changes enable the MyCustomers application to use the modified stored procedure, and no changes need to be made to the application. The changes add new possibilities for future applications because the modified stored procedure accepts the order year as a parameter.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Pass Multiple Parameters to the Stored Procedure

1. Highlight the following T-SQL code under the task 3 description:

```
ALTER PROCEDURE Sales.GetTopCustomers
@orderyear int = NULL,
@n int = 10
AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE YEAR(o.orderdate) = @orderyear OR @orderyear IS NULL
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT @n ROWS ONLY;
```

2. Click **Execute**. You have modified the Sales.GetTopCustomers stored procedure to have an additional parameter named @n. You can use this parameter to specify how many customers to retrieve. The default value is 10.
3. After the previous T-SQL code, type the following T-SQL code to execute the modified stored procedure:

```
EXECUTE Sales.GetTopCustomers;
```

4. Highlight the written query and click **Execute**.
5. After the previous T-SQL code, type the following T-SQL code to retrieve the top five customers for the year 2008:

```
EXECUTE Sales.GetTopCustomers @orderyear = 2008, @n = 5;
```

6. Highlight the written query and click **Execute**.
7. After the previous T-SQL code, type the following T-SQL code to retrieve the top 10 customers for the year 2007:

```
EXECUTE Sales.GetTopCustomers @orderyear = 2007;
```

8. Highlight the written query and click **Execute**.
9. After the previous T-SQL code, type the following T-SQL code to retrieve the top 20 customers:

```
EXECUTE Sales.GetTopCustomers @n = 20;
```

10. Highlight the written query and click **Execute**.
11. Do the applications using the stored procedure need to be changed because another parameter was added? No changes need to be made to the application.

MCT USE ONLY, STUDENT USE PROHIBITED

▶ Task 4: Return the Result from a Stored Procedure Using the OUTPUT Clause

1. Highlight the following T-SQL code under the task 4 description:

```
ALTER PROCEDURE Sales.GetTopCustomers
@customerpos int = 1,
@customername nvarchar(30) OUTPUT
AS
SET @customername = (
SELECT
c.contactname
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
GROUP BY c.custid, c.contactname
ORDER BY SUM(o.val) DESC
OFFSET @customerpos - 1 ROWS FETCH NEXT 1 ROW ONLY
);
```

2. Click **Execute**.
3. Find the following DECLARE statement in the provided code:

```
DECLARE @outcustomername nvarchar(30);
```

   This statement declares a parameter named @outcustomername.

4. After the DECLARE statement, add code that uses the OUTPUT clause to return the stored procedure's result as a variable named @outcustomername. Your code, together with the provided DECLARE statement, should look like this:

```
DECLARE @outcustomername nvarchar(30);
EXECUTE Sales.GetTopCustomers @customerpos = 1, @customername = @outcustomername
OUTPUT;
SELECT @outcustomername AS customername;
```

5. Highlight all three T-SQL statements and click **Execute**.

Results: After this exercise, you should know how to invoke stored procedures that have parameters.

Exercise 3: Executing System Stored Procedures

▶ Task 1: Execute the Stored Procedure sys.sp_help

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following T-SQL code after the task 1 description:

```
EXEC sys.sp_help;
```

4. Highlight the written query and click **Execute**.
5. In the query pane, type the following T-SQL code after the previous T-SQL code:

```
EXEC sys.sp_help N'Sales.Customers';
```

6. Highlight the written query and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Execute the Stored Procedure sys.sp_helptext

1. In the query pane, type the following T-SQL code after the task 2 description:

```
EXEC sys.sp_helptext N'Sales.GetTopCustomers';
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Execute the Stored Procedure sys.sp_columns

1. In the query pane, type the following T-SQL code after the task 3 description:

```
EXEC sys.sp_columns @table_name = N'Customers', @table_owner = N'Sales';
```

2. Highlight the written query and click **Execute**.

▶ Task 4: Drop the Created Stored Procedure

1. Highlight the provided T-SQL statement under the task 4 description and click **Execute**.

Results: After this exercise, you should have a basic knowledge of invoking different system-stored procedures.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 16: Programming with T-SQL

Lab: Programming with T-SQL

Exercise 1: Declaring Variables and Delimiting Batches

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab16\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Declare a Variable and Retrieve the Value

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab16\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and then double-click the query **51 - Lab Exercise 1.sql** (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard).
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following T-SQL code after the task 1 description:

```
DECLARE @num int = 5;
SELECT @num AS mynumber;
```

7. Highlight the written T-SQL code and click **Execute**.
8. In the query pane, type the following T-SQL code after the previous one:

```
DECLARE
@num1 int,
@num2 int;
SET @num1 = 4;
SET @num2 = 6;
SELECT @num1 + @num2 AS totalnum;
```

9. Highlight the written T-SQL code and click **Execute**.

▶ Task 3: Set the Variable Value Using a SELECT Statement

1. In the query pane, type the following T-SQL code after the task 2 description:

```
DECLARE @empname nvarchar(30);
SET @empname = (SELECT firstname + N' ' + lastname FROM HR.Employees WHERE empid =
1);
SELECT @empname AS employee;
```

2. Highlight the written T-SQL code and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

3. Observe the result. What would happen if the SELECT statement was returning more than one row? You would get an error because the SET statement requires you to use a scalar subquery to pull data from a table. Remember that a scalar subquery fails at runtime if it returns more than one value.

▶ Task 4: Use a Variable in the WHERE Clause

1. In the query pane, type the following T-SQL code after the task 3 description:

```
DECLARE
@empname nvarchar(30),
@empid int;
SET @empid = 5;
SET @empname = (SELECT firstname + N' ' + lastname FROM HR.Employees WHERE empid =
@empid);
SELECT @empname AS employee;
```

2. Highlight the written T-SQL code and click **Execute**.
3. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab16\Solution\55 - Lab Exercise 1 - Task 3 Result.txt.
4. Change the @empid variable's value from 5 to 2 and execute the modified T-SQL code to observe the changes.

▶ Task 5: Use Variables with Batches

1. Highlight the T-SQL code in task 4. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 4 description. On the toolbar, click **Edit** and then **Paste**.
3. In the code you just copied, add the batch delimiter GO before this statement:

```
SELECT @empname AS employee;
```

4. Make sure your T-SQL code looks like this:

```
DECLARE
@empname nvarchar(30),
@empid int;
SET @empid = 5;
SET @empname = (SELECT firstname + N' ' + lastname FROM HR.Employees WHERE empid =
@empid)
GO
SELECT @empname AS employee;
```

5. Highlight the written T-SQL code and click **Execute**.
6. Observe the error:

```
Must declare the scalar variable "@empname".
```

7. Can you explain why the batch delimiter caused an error? Variables are local to the batch in which they are defined. If you try to refer to a variable that was defined in another batch, you get an error saying that the variable was not defined. Also, keep in mind that GO is a client command, not a server T-SQL command.

Results: After this exercise, you should know how to declare and use variables in T-SQL code.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Using Control-of-Flow Elements

▶ Task 1: Write Basic Conditional Logic

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following T-SQL code after the task 1 description:

```
DECLARE
@i int = 8,
@result nvarchar(20);
IF @i < 5
SET @result = N'Less than 5'
ELSE IF @i <= 10
SET @result = N'Between 5 and 10'
ELSE if @i > 10
SET @result = N'More than 10'
ELSE
SET @result = N'Unknown';
SELECT @result AS result;
```

4. Highlight the written T-SQL code and click **Execute**.
5. In the query pane, type the following T-SQL code:

```
DECLARE
@i int = 8,
@result nvarchar(20);
SET @result =
CASE
WHEN @i < 5 THEN
N'Less than 5'
WHEN @i <= 10 THEN
N'Between 5 and 10'
WHEN @i > 10 THEN
N'More than 10'
ELSE
N'Unknown'
END;
SELECT @result AS result;
```

   This code uses a CASE expression and only one SET expression to get the same result as the previous T-SQL code. Remember to use a CASE expression when it is a matter of returning an expression. However, if you need to execute multiple statements, you cannot replace IF with CASE.

6. Highlight the written T-SQL code and click **Execute**.

▶ Task 2: Check the Employee Birthdate

1. In the query pane, type the following T-SQL code after the task 2 description:

```
DECLARE
@birthdate date,
@cmpdate date;
SET @birthdate = (SELECT birthdate FROM HR.Employees WHERE empid = 5);
SET @cmpdate = '19700101';
IF @birthdate < @cmpdate
PRINT 'The person selected was born before January 1, 1970'
ELSE
PRINT 'The person selected was born on or after January 1, 1970';
```

2. Highlight the written T-SQL code and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Create and Execute a Stored Procedure

1. Highlight the following T-SQL code under the task 3 description:

```
CREATE PROCEDURE Sales.CheckPersonBirthDate
@empid int,
@cmpdate date
AS
DECLARE
@birthdate date;
SET @birthdate = (SELECT birthdate FROM HR.Employees WHERE empid = @empid);
IF @birthdate < @cmpdate
PRINT 'The person selected was born before ' + FORMAT(@cmpdate, 'MMMM d, yyyy', 'en-
US');
ELSE
PRINT 'The person selected was born on or after ' + FORMAT(@cmpdate, 'MMMM d, yyyy',
'en-US');
```

2. Click **Execute**. You have created a stored procedure named Sales.CheckPersonBirthDate. It has two parameters: @empid, which you use to specify an employee ID, and @cmpdate, which you use as a comparison date.
3. In the query pane, type the following T-SQL code after the provided T-SQL code:

```
EXECUTE Sales.CheckPersonBirthDate @empid = 3, @cmpdate = '19900101';
```

4. Highlight the written T-SQL code and click **Execute**.

▶ Task 4: Execute a Loop Using the WHILE Statement

1. In the query pane, type the following T-SQL code after the task 4 description:

```
DECLARE @i int = 1;
WHILE @i <= 10
BEGIN
PRINT @i;
SET @i = @i + 1;
END;
```

2. Highlight the written T-SQL code and click **Execute**.

▶ Task 5: Remove the Stored Procedure

1. Highlight the following T-SQL code under the task 5 description:

```
DROP PROCEDURE Sales.CheckPersonBirthDate;
```

2. Click **Execute**.

Results: After this exercise, you should know how to control the flow of the elements inside the T-SQL code.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Using Variables in a Dynamic SQL Statement

▶ Task 1: Write a Dynamic SQL Statement That Does Not Use a Parameter

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following T-SQL code after the task 1 description:

```
DECLARE @SQLstr nvarchar(200);
SET @SQLstr = N'SELECT empid, firstname, lastname FROM HR.Employees';
EXECUTE sys.sp_executesql @statement = @SQLstr;
```

4. Highlight the written T-SQL code and click **Execute**.

▶ Task 2: Write a Dynamic SQL Statement That Uses a Parameter

1. Highlight the T-SQL code in task 1. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 2 description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the T-SQL code to look like this:

```
DECLARE
@SQLstr nvarchar(200),
@SQLparam nvarchar(100);
SET @SQLstr = N'SELECT empid, firstname, lastname FROM HR.Employees WHERE empid =
@empid';
SET @SQLparam = N'@empid int';
EXECUTE sys.sp_executesql @statement = @SQLstr, @params = @SQLparam, @empid = 5;
```

4. Highlight the written T-SQL code and click **Execute**.

Results: After this exercise, you should have a basic knowledge of generating and invoking dynamic SQL statements.

Exercise 4: Using Synonyms

▶ Task 1: Create and Use a Synonym for a Table

1. In Solution Explorer, double-click the query **81 - Lab Exercise 4.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following T-SQL code after the task 1 description:

```
CREATE SYNONYM dbo.Person
FOR AdventureWorks.Person.Person;
```

4. Highlight the written T-SQL code and click **Execute**. You have created a synonym named dbo.Person.
5. In the query pane, type the following SELECT statement after the previous T-SQL code:

```
SELECT FirstName, LastName
FROM dbo.Person;
```

6. Highlight the written query and click **Execute**.

▶ Task 2: Drop the Synonym

1. Highlight the following T-SQL code under the task 2 description:

```
DROP SYNONYM dbo.Person;
```

2. Click **Execute**.

Results: After this exercise, you should know how to create and use a synonym.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 17: Implementing Error Handling

Lab: Implementing Error Handling

Exercise 1: Redirecting Errors with TRY/CATCH

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab17\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Write a Basic TRY/CATCH Construct

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab17\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand **Queries** and then double-click the **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the View menu or press Ctrl+Alt+L on the keyboard).
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. Highlight the following SELECT statement under the task 1 description:

```
SELECT CAST(N'Some text' AS int);
```

7. Click **Execute**. Notice the conversion error.
8. Write a TRY/CATCH construct. Your T-SQL code should look like this:

```
BEGIN TRY
SELECT CAST(N'Some text' AS int);
END TRY
BEGIN CATCH
PRINT 'Error';
END CATCH;
```

9. Highlight the written T-SQL code and click **Execute**.

▶ Task 3: Display an Error Number and an Error Message

1. Highlight the following T-SQL code under the task 2 description:

```
DECLARE @num varchar(20) = '0';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
END CATCH;
```

2. Click **Execute**. Notice that you did not get an error because you used the TRY/CATCH construct.

MCT USE ONLY. STUDENT USE PROHIBITED

3. Modify the T-SQL code by adding two PRINT statements. The T-SQL code should look like this:

```
DECLARE @num varchar(20) = '0';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS varchar(10));
PRINT 'Error Message: ' + ERROR_MESSAGE();
END CATCH;
```

4. Highlight the T-SQL code and click **Execute**.
5. Change the value of the @num variable to look like this:

```
DECLARE @num varchar(20) = 'A';
```

6. Highlight the T-SQL code and click **Execute**. Notice that you get a different error number and message.
7. Change the value of the @num variable to look like this:

```
DECLARE @num varchar(20) = ' 1000000000';
```

8. Highlight the T-SQL code and click **Execute**. Notice that you get a different error number and message.

▶ Task 4: Add Conditional Logic to a CATCH Block

1. Modify the T-SQL code in Task 3 to look like this:

```
DECLARE @num varchar(20) = 'A';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
IF ERROR_NUMBER() IN (245, 8114)
BEGIN
PRINT 'Handling conversion error...'
END
ELSE
BEGIN
PRINT 'Handling non-conversion error...';
END;
PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS varchar(10));
PRINT 'Error Message: ' + ERROR_MESSAGE();
END CATCH;
```

2. Highlight the written query and click **Execute**.
3. Change the value of the @num variable to look like this:

```
DECLARE @num varchar(20) = '0';
```

4. Highlight the T-SQL code and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 5: Execute a Stored Procedure in the CATCH Block

1. Highlight the following T-SQL code under the task 4 description:

```
CREATE PROCEDURE dbo.GetErrorInfo AS
PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS varchar(10));
PRINT 'Error Message: ' + ERROR_MESSAGE();
PRINT 'Error Severity: ' + CAST(ERROR_SEVERITY() AS varchar(10));
PRINT 'Error State: ' + CAST(ERROR_STATE() AS varchar(10));
 PRINT 'Error Line: ' + CAST(ERROR_LINE() AS varchar(10));
 PRINT 'Error Proc: ' + COALESCE(ERROR_PROCEDURE(), 'Not within procedure');
```

2. Click **Execute**. You have created a stored procedure named dbo.GetErrorInfo.
3. Modify the T-SQL code under **TRY/CATCH** to look like this:

```
DECLARE @num varchar(20) = '0';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
EXECUTE dbo.GetErrorInfo;
END CATCH;
```

4. Highlight the written T-SQL code and click **Execute**.

Exercise 2: Using THROW to Pass an Error Message Back to a Client

▶ Task 1: Re-Throw the Existing Error Back to a Client

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. Modify the T-SQL code under the Task 1 description to look like this:

```
DECLARE @num varchar(20) = '0';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
EXECUTE dbo.GetErrorInfo; THROW;
END CATCH;
```

4. Highlight the written T-SQL code and click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY STUDENT USE PROHIBITED

▶ Task 2: Add an Error Handling Routine

1. Modify the T-SQL code under the Task 2 description to look like this:

```
DECLARE @num varchar(20) = 'A';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
EXECUTE dbo.GetErrorInfo;
IF ERROR_NUMBER() = 8134
BEGIN
PRINT 'Handling devision by zero...';
END
ELSE
BEGIN
PRINT 'Throwing original error';
THROW;
END;
END CATCH;
```

2. Highlight the written T-SQL code and click **Execute**.

▶ Task 3: Add a Different Error Handling Routine

1. Find the following T-SQL code under the task 3 description:

```
DECLARE @msg AS varchar(2048);
SET @msg = 'You are doing the exercise for Module 17 on ' + FORMAT(CURRENT_TIMESTAMP,
'MMMM d, yyyy', 'en-US') + '. It''s not an error but it means that you are near the
final module!';
```

2. After the provided code, add a THROW statement. The completed T-SQL code should look like this:

```
DECLARE @msg AS varchar(2048);
SET @msg = 'You are doing the exercise for Module 17 on ' + FORMAT(CURRENT_TIMESTAMP,
'MMMM d, yyyy', 'en-US') + '. It''s not an error but it means that you are near the
final module!';
THROW 50001, @msg, 1;
```

3. Highlight the written T-SQL code and click **Execute**.

▶ Task 4: Remove the Stored Procedure

1. Highlight the provided T-SQL statement under the task 4 description and click **Execute**.

Results: After this exercise, you should know how to throw an error to pass messages back to a client.

Module 18: Implementing Transactions

Lab: Implementing Transactions

Exercise 1: Controlling Transactions with BEGIN, COMMIT, and ROLLBACK

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab18\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Commit a Transaction

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab18\Starter\Project\Project.ssmssln**.
4. In **Solution Explorer**, double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard).
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. Modify the T-SQL code under the task 1 description by adding the BEGIN TRAN and COMMIT TRAN statements. Your T-SQL code should look like this:

```
BEGIN TRAN;
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'20110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
COMMIT TRAN;
```

7. Highlight the written T-SQL code and click **Execute**.
8. In the query pane, type the following query after the previous T-SQL code:

```
SELECT empid, lastname, firstname
FROM HR.Employees
ORDER BY empid DESC;
```

9. Highlight the written query and click **Execute**.

▶ Task 3: Delete the Previously Inserted Rows from the HR.Employees Table

1. Highlight the following T-SQL code under the task 2 description:

```
DELETE HR.Employees
WHERE empid IN (10, 11);
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

MCT USE ONLY. STUDENT USE PROHIBITED

2. Click **Execute**.

▶ Task 4: Open a Transaction and Use the ROLLBACK Statement

1. Modify the T-SQL code under the task 3 description by adding the BEGIN TRAN statement. Your T-SQL code should look like this:

```
BEGIN TRAN;
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'20110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
```

2. Highlight the written T-SQL code and click **Execute**.
3. In the query pane, type the following query after the previous T-SQL code:

```
SELECT empid, lastname, firstname
FROM HR.Employees
ORDER BY empid DESC;
```

4. Highlight the written query and click **Execute**.
5. In the query pane, type the following statement after the SELECT statement:

```
ROLLBACK TRAN;
```

6. Highlight the written statement and click **Execute**.
7. Again, highlight the SELECT statement shown in step 3 and click **Execute**.

▶ Task 5: Clear the Modifications Against the HR.Employees Table

1. Highlight the following T-SQL code after the task 4 description:

```
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

2. Click **Execute**.

Results: After this exercise, you should be able to control a transaction using the BEGIN TRAN, COMMIT, and ROLLBACK statements.

Exercise 2: Adding Error Handling to a CATCH Block

▶ Task 1: Observe the Provided T-SQL Code

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. Highlight only the following SELECT statement under the task 1 description:

```
SELECT empid, lastname, firstname
FROM HR.Employees
ORDER BY empid DESC;
```

MCT USE ONLY. STUDENT USE PROHIBITED

4. Click **Execute**.
5. In the provided T-SQL code, highlight the code between the BEGIN TRAN and COMMIT TRAN statements. Your highlighted T-SQL code should look like this:

```
BEGIN TRAN;
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'10110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
COMMIT TRAN;
```

6. Click **Execute**. Notice there is a conversion error in the second INSERT statement.
7. Again, highlight the SELECT statement shown in step 3 and click **Execute**.

▶ Task 2: Delete the Previously Inserted Row in the HR.Employees Table

1. Highlight the following T-SQL code under the task 2 description:

```
DELETE HR.Employees
WHERE empid IN (10, 11);
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

2. Click **Execute**.

▶ Task 3: Abort Both INSERT Statements If an Error Occurs

1. Modify the T-SQL code under the task 3 description to look like this:

```
BEGIN TRY
BEGIN TRAN;
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'10110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
PRINT 'Commit the transaction...';
COMMIT TRAN;
END TRY
BEGIN CATCH
IF @@TRANCOUNT > 0
BEGIN
PRINT 'Rollback the transaction...';
ROLLBACK TRAN;
END
END CATCH;
```

2. Highlight the modified T-SQL code and click **Execute**.
3. In the query pane, type the following query after the modified T-SQL code:

```
SELECT empid, lastname, firstname
FROM HR.Employees ORDER BY empid DESC;
```

MCT USE ONLY. STUDENT USE PROHIBITED

4. Highlight the written query and click **Execute**.

▶ Task 4: Clear the Modifications Against the HR.Employees Table

1. Highlight the following T-SQL code under the task 4 description:

```
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

2. Click **Execute**.

Results: After this exercise, you should have a basic understanding of how to control a transaction inside a TRY/CATCH block to efficiently handle possible errors.

MCT USE ONLY. STUDENT USE PROHIBITED

Appendix A: Improving Query Performance

Lab: Improving Query Performance

Exercise 1: Viewing Query Execution Plans

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab19\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Create and populate the sample table Sales.TempOrders

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab19\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard.)
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, highlight the T-SQL code after the task 1 description and click **Execute**.

▶ Task 3: Show estimated and actual execution plans

1. In the query pane, type the following query after the task 2 description:

```
SELECT orderid, custid, orderdate
FROM Sales.TempOrders;
```

2. Highlight the written query and click **Display Estimated Execution Plan**.
3. In the **Results** pane, click the **Execution plan** tab. Hover your mouse pointer over the Table Scan operator and look at the properties displayed in the yellow tooltip box.
4. Position your mouse pointer over the arrow between the SELECT operator and the Table Scan operator in the execution plan. You should see three properties: Estimated Number of Rows, Estimated Data Size, and Estimated Row Size.
5. Right-click the **SELECT** operator and click **Properties** in the context menu.
6. Click the SELECT operator.
7. On the toolbar, click **Include Actual Execution Plan**.
8. Highlight the written query and click **Execute**.
9. In the **Results** pane, click the **Execution plan** tab and observe the actual execution plan.

MCT USE ONLY, STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Analyze the execution plan of another SELECT statement

1. Highlight the previous query in task 2. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 3 description. On the toolbar, click **Edit** and then **Paste**.
3. In the query pane, alter the copied query to look like this:

```
SELECT TOP (1) orderid, custid, orderdate
FROM Sales.TempOrders;
```

4. Highlight the altered query and click **Display Estimated Execution Plan**.
5. Compare this task's execution plan with the one for the previous task. Which operator is new? The TOP operator is new.

▶ Task 5: Graphically compare two execution plans

1. Highlight the query in task 2. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 4 description. On the toolbar, click **Edit** and then **Paste**.
3. Highlight the query in task 3. On the toolbar, click **Edit** and then **Copy**.
4. In the query window, click the line after the copied SELECT statement. On the toolbar, click **Edit** and then **Paste**.
5. Highlight both SELECT statements and click **Display Estimated Execution Plan**.
6. In the toolbar, click **Include Actual Execution Plan**.
7. Highlight both SELECT statements and click **Execute**.

Results: After this exercise, you should be able to display estimated and actual execution plans.

Exercise 2: Viewing Index Usage and Using SET STATISTICS Statements

▶ Task 1: Create a clustered index and write a SELECT statement

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. Highlight the provided T-SQL code after the task 1 description and click **Execute**.
4. In the query pane, type the following query after the provided T-SQL code:

```
SELECT orderid, custid, orderdate
FROM Sales.TempOrders
WHERE YEAR(orderdate) = 2007 AND MONTH(orderdate) = 6;
```

5. Highlight the written query and click **Execute**.
6. Highlight the written query and click **Display Estimated Execution Plan**.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Enable I/O statistics to observe the number of needed reads

1. In the query pane, type the following T-SQL statement after the task 2 description:

```
SET STATISTICS IO ON;
```

2. Highlight the written statement and click **Execute**.
3. Highlight the query in task 1. On the toolbar, click **Edit** and then **Copy**.
4. In the query window, click the line after the written T-SQL statement. On the toolbar, click **Edit** and then **Paste**.
5. Highlight the copied SELECT statement and click **Execute**.
6. In the **Results** pane, click the **Messages** tab and observe the number of logical reads.

▶ Task 3: Modify the SELECT statement to use a search argument in the WHERE clause

1. Highlight the SELECT statement in task 1. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 3 description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the SELECT statement to look like this:

```
SELECT orderid, custid, orderdate
FROM Sales.TempOrders
WHERE orderdate >= '20070601' AND orderdate < '20070701';
```

4. Highlight the modified query and click **Execute**.
5. Highlight the modified query and click **Display Estimated Execution Plan**.

▶ Task 4: Compare both SELECT statements

1. Highlight the SELECT statement in task 1. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 4 description. On the toolbar, click **Edit** and then **Paste**.
3. Highlight the query in task 3. On the toolbar, click **Edit** and then **Copy**.
4. In the query window, click the line after the copied SELECT statement. On the toolbar, click **Edit** and then **Paste**.
5. Highlight both SELECT statements and click **Execute**.
6. Highlight both SELECT statements and click **Display Estimated Execution Plan**.
7. Compare the execution plans for the two queries. Why is the SELECT statement from task 3 so much faster? This SELECT statement efficiently uses the created clustered index and does a clustered index seek operation. The SELECT statement from task 1 does a clustered index scan (that is, a table scan).

▶ Task 5: Remove the created table and disable I/O statistics

1. Highlight the provided T-SQL code after the Task 5 description and click **Execute**.

Results: After this exercise, you should have a basic understanding of how to enable SET STATISTICS options. Remember to invest time in understanding indexes so that you can write efficient queries.

MCT USE ONLY. STUDENT USE PROHIBITED

Appendix B: Querying SQL Server Metadata

Lab: Querying SQL Server Metadata

Exercise 1: Querying System Catalog Views

▶ Task 1: Prepare the Lab Environment

1. Ensure that the 20461C-MIA-DC and 20461C-MIA-SQL virtual machines are both running, and then log on to 20461C-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab20\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Write a SELECT statement to retrieve all databases

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab20\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand Queries and double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard.)
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
SELECT name, dbid, crdate
FROM sys.sysdatabases;
```

7. Highlight the written query and click **Execute**. Observe that the query retrieves a row for each database.

▶ Task 3: Write a SELECT statement to retrieve all user-defined tables in the TSQL database

1. In the query pane, type the following query after the task 2 description:

```
SELECT
object_id, name, schema_id, type, type_desc, create_date, modify_date
FROM sys.objects;
```

2. Highlight the written query and click **Execute**.
3. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
4. In the query window, click the line after the written T-SQL statement. On the toolbar, click **Edit** and then **Paste**.
5. Modify the T-SQL statement to retrieve all distinct values for the columns **type** and **type_desc**. The query should look like this:

```
SELECT DISTINCT
type, type_desc
FROM sys.objects
ORDER BY type_desc;
```

MCT USE ONLY. STUDENT USE PROHIBITED

6. Highlight the written query and click **Execute**.
7. Highlight the first query. On the toolbar, click **Edit** and then **Copy**.
8. In the query window, click the line after the written T-SQL statement. On the toolbar, click **Edit** and then **Paste**.
9. Modify the T-SQL statement to filter only user-defined tables. The query should look like this:

```
SELECT
object_id, name, schema_id, type, type_desc, create_date, modify_date
FROM sys.objects
WHERE type = N'U';
```

10. Highlight the written query and click **Execute**.

▶ Task 4: Use a different approach to retrieve all user-defined tables in the TSQL database

1. In the query pane, type the following query after the task 3 description:

```
SELECT
object_id, name, SCHEMA_NAME(schema_id) AS schemaname, type, type_desc, create_date,
modify_date
FROM sys.tables;
```

2. Highlight the written query and click **Execute**.
3. In the query pane, type the following query after the previous one:

```
SELECT
object_id, name, SCHEMA_NAME(schema_id) AS schemaname, type, type_desc, create_date,
modify_date
FROM sys.views;
```

4. Highlight the written query and click **Execute**.

▶ Task 5: Write a SELECT statement to retrieve all columns from the Sales.Customers table

1. In the query pane, type the following query after the task 4 description:

```
SELECT
c.name AS columnname, c.column_id, c.system_type_id, c.max_length, c.precision,
c.scale, c.collation_name
FROM sys.columns AS c
WHERE object_id = OBJECT_ID('Sales.Customers')
ORDER BY c.column_id;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should be able to retrieve some system information from the system catalog views.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Querying System Functions

▶ Task 1: Write a SELECT statement to retrieve the current database name

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
DB_ID() AS databaseid,
DB_NAME(DB_ID()) AS databasename,
USER_NAME() as currusername;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Write a SELECT statement to retrieve the object name and schema name

1. In the query pane, type the following query after the task 2 description:

```
SELECT
name,
OBJECT_NAME(object_id) AS tablename,
OBJECT_SCHEMA_NAME(object_id) AS schemaname
FROM sys.columns;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT statement to retrieve all the columns from the user-defined tables that contain the word "name" in the column name

1. In the query pane, type the following query after the task 3 description:

```
SELECT
c.name AS columnname,
OBJECT_NAME (c.object_id) AS tablename,
OBJECT_SCHEMA_NAME(c.object_id) AS schemaname
FROM sys.columns AS c
WHERE
c.name LIKE N'%name%'
AND OBJECTPROPERTY(c.object_id, N'IsUserTable') = 1;
```

2. Highlight the written query and click **Execute**.

▶ Task 4: Retrieve the view definition

1. In the query pane, type the following query after the task 4 description:

```
SELECT OBJECT_DEFINITION(OBJECT_ID(N'Sales.CustOrders'));
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should know how to use different system functions.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Querying System Dynamic Management Views

▶ Task 1: Write a SELECT statement to return all current sessions

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
session_id, login_time, host_name, language, date_format
FROM
sys.dm_exec_sessions;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Execute the provided T-SQL statement

1. Highlight the following T-SQL code under the task 2 description:

```
SELECT
cpu_count AS 'Logical CPU Count',
hyperthread_ratio AS 'Hyperthread Ratio',
cpu_count/hyperthread_ratio As 'Physical CPU Count',
physical_memory_kb/1024 AS 'Physical Memory (MB)',
sqlserver_start_time AS 'Last SQL Start'
FROM sys.dm_os_sys_info;
```

2. Click **Execute**.

▶ Task 3: Write a SELECT statement to retrieve the current memory information

1. In the query pane, type the following query after the task 3 description:

```
SELECT
total_physical_memory_kb,
available_physical_memory_kb,
total_page_file_kb,
available_page_file_kb,
system_memory_state_desc
FROM sys.dm_os_sys_memory;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should have an understanding of how to write queries against the system DMVs.

MCT USE ONLY. STUDENT USE PROHIBITED