

OFFICIAL MICROSOFT LEARNING PRODUCT

20761B

Querying Data with Transact-SQL

STUDENT 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.

© 2017 Microsoft Corporation. All rights reserved.

Microsoft and the trademarks listed at <https://www.microsoft.com/en-us/legal/intellectualproperty/trademarks/en-us.aspx> are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

Product Number: 20761B

Part Number (if applicable): X21-31969

Released: 02/2017

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.

- I. "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,
- b. **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,

- 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.

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.

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.

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.

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

Cette limitation concerne:

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

Elle s'applique également, même si Microsoft 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

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 performance¹. 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



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

Acknowledgements

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

Aaron Johal – Content Developer

Aaron Johal is a Microsoft Certified Trainer who splits his time between training, consultancy, content development, contracting and learning. Since he moved into the non-functional side of the Information Technology business. He has presented technical sessions at SQL Pass in Denver and at sqlbits in London. He has also taught and worked in a consulting capacity throughout the UK and abroad, including Africa, Spain, Saudi Arabia, Netherlands, France, and Ireland. He enjoys interfacing functional and non-functional roles to try and close the gaps between effective use of Information Technology and the needs of the Business.

Caroline Eveleigh – Content Developer

Caroline Eveleigh is a Microsoft Certified Professional and SQL Server specialist. She has worked with SQL Server since version 6.5 and, before that, with Microsoft Access and dBase. Caroline works on database development and Microsoft Azure projects for both corporates, and small businesses. She is an experienced business analyst, helping customers to re-engineer business processes, and improve decision making using data analysis. Caroline is a trained technical author and a frequent blogger on project management, business intelligence, and business efficiency. Between development projects, Caroline is a keen SQL Server evangelist, speaking and training on SQL Server and Azure SQL Database.

Ed Harper – Content Developer

Ed Harper is a database developer specializing in Microsoft SQL Server. Ed has worked with SQL Server since 1999, and has developed and designed transaction-processing and reporting systems for cloud security, telecommunications, and financial services companies.

Jamie Newman – Content Developer

Jamie Newman became an IT trainer in 1997, first for an IT training company and later for a university, where he became involved in developing courses as well as training them. He began to specialize in databases and eventually moved into database consultancy. In recent years he has specialized in SQL Server and has set up multi user systems that are accessed nationwide. Despite now being more involved with development work, Jamie still likes to deliver IT training courses when the opportunity arises!

John Daisley – Content Developer

John Daisley is a mixed vendor Business Intelligence and Data Warehousing contractor with a wealth of data warehousing and Microsoft SQL Server database administration experience. Having worked in the Business Intelligence arena for over a decade, John has extensive experience of implementing business intelligence and database solutions using a wide range of technologies and across a wide range of industries including airlines, engineering, financial services, and manufacturing.

Nick Anderson – Content Developer

Nick Anderson MBCS MISTC has been a freelance Technical Writer since 1987 and Trainer since 1999. Nick has written internal and external facing content in many business and technical areas including development, infrastructure and finance projects involving SQL Server, Visual Studio and similar tools. Nick provides services for both new and existing document processes from knowledge capture to publishing.

Phil Stollery – Content Developer

Phil has been providing IT consultancy to South West England since graduating in Computer Science. He has worked with small and large organizations to improve their use of SQL Server, predominantly focusing on business information and surrounding technologies such as SharePoint. Most recently, Phil worked with the National Health Service in Gloucestershire on a custom intranet built on SharePoint. A trusted partner, he can communicate at all levels, from technical staff to senior management. Phil brings a wealth of experience that enhances any project.

Rachel Horder – Content Developer

Rachel Horder graduated with a degree in Journalism and began her career in London writing for The Times technology supplement. After discovering a love for programming, Rachel became a full-time developer, and now provides SQL Server consultancy services to businesses across a wide variety of industries. Rachel is MCSA certified, and continues to write technical articles and books, including What's New in SQL Server 2012. As an active member of the SQL Server community, Rachel organizes the Bristol SQL Server Club user group, runs the Bristol leg of SQL Relay, and is a volunteer at SQLBits.

Simon Butler – Content Developer

Simon Butler FISTC is a highly-experienced Senior Technical Writer with nearly 30 years' experience in the profession. He has written training materials and other information products for several high-profile clients. He is a Fellow of the Institute of Scientific and Technical Communicators (ISTC), the UK professional body for Technical Writers/Authors. To gain this, his skills, experience and knowledge have been judged and assessed by the Membership Panel. He is also a Past President of the Institute and has been a tutor on the ISTC Open Learning course in Technical Communication techniques. His writing skills are augmented by extensive technical skills gained within the computing and electronics fields.

Geoff Allix – Technical Reviewer

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.

Lin Joyner – Technical Reviewer

Lin is an experienced Microsoft SQL Server developer and administrator. She has worked with SQL Server since version 6.0 and previously as a Microsoft Certified Trainer, delivered training courses across the UK. Lin has a wide breadth of knowledge across SQL Server technologies, including BI and Reporting Services. Lin also designs and authors SQL Server and .NET development training materials. She has been writing instructional content for Microsoft for over 15 years.

Contents

Module 1: Introduction to Microsoft SQL Server 2016

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	1-9
Lab: Working with SQL Server 2016 Tools	1-16
Module Review and Takeaways	1-19

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	2-17
Lesson 4: Understanding the Logical Order of Operations in SELECT Statements	2-19
Lab: Introduction to T-SQL Querying	2-24
Module Review and Takeaways	2-27

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

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	5-20
Module Review and Takeaways	5-25

Module 6: Working with SQL Server 2016 Data Types

Module Overview	6-1
Lesson 1: Introducing SQL Server 2016 Data Types	6-2
Lesson 2: Working with Character Data	6-11
Lesson 3: Working with Date and Time Data	6-20
Lab: Working with SQL Server 2016 Data Types	6-26
Module Review and Takeaways	6-32

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	7-8
Lesson 3: Generating Automatic Column Values	7-12
Lab: Using DML to Modify Data	7-15
Module Review and Takeaways	7-18

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	8-16
Lab: Using Built-in Functions	8-20
Module Review and Takeaways	8-24

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	9-15
Lab: Grouping and Aggregating Data	9-18
Module Review and Takeaways	9-24

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	10-10
Lab: Using Subqueries	10-13
Module Review and Takeaways	10-18

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	12-16
Module Review and Takeaways	12-21

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	14-12
Module Review and Takeaways	14-18

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	15-15
Module Review and Takeaways	15-21

Module 16: Programming with T-SQL

Module Overview	16-1
Lesson 1: T-SQL Programming Elements	16-2
Lesson 2: Controlling Program Flow	16-8
Lab: Programming with T-SQL	16-12
Module Review and Takeaways	16-18

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	17-15

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

Lab Answer Keys

Module 1 Lab: Working with SQL Server 2016 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 2016 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
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

About This Course

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

Course Description

 **Note:** This release ('B') MOC version of course 20761 has been developed on RTM software. This version supersedes 20761A.

The main purpose of this 5 day instructor led course is to give students a good understanding of the Transact-SQL language which is used by all SQL Server-related disciplines; namely, Database Administration, Database Development and Business Intelligence. As such, the primary target audience for this course is: Database Administrators, Database Developers and BI professionals.

The course will very likely be well attended by SQL power users who aren't necessarily database-focused; namely, report writers, business analysts and client application developers.

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; namely, report writers, business analysts and client application developers.

Student Prerequisites

This course requires that you meet the following prerequisites:

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

Course Objectives

After completing this course, students will be able to:

- Describe the basic architecture and concepts of Microsoft SQL Server 2016
- Understand the similarities and differences between Transact-SQL and other computer languages
- Write SELECT queries
- Query multiple tables
- Sort and filter data
- Describe the use of data types in SQL Server
- Modify data using Transact-SQL
- Use built-in functions
- Group and aggregate data
- Use subqueries
- Use table expressions
- Use set operators
- Use window ranking, offset and aggregate functions

MCT USE ONLY STUDENT USE PROHIBITED

- Implement pivoting and grouping sets
- Execute stored procedures
- Program with T-SQL
- Implement error handling
- Implement transactions

Course Outline

The course outline is as follows:

Module 1, "Introduction to Microsoft SQL Server 2016"

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 2016 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 Windows Ranking, Offset, and Aggregate Functions"

Module 14, "Pivoting and Grouping Sets"

Module 15, "Executing Stored Procedures"

Module 16, "Programming with T-SQL"

Course Materials

The following materials are included with your kit:

- **Course Handbook:** a succinct classroom learning guide that provides the critical technical information in a crisp, tightly-focused format, which is essential 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 on-the-job reference material to boost knowledge and skills retention.
 - **Lab Answer Keys:** provide step-by-step lab solution guidance.



Additional Reading: Course Companion Content on the <http://www.microsoft.com/learning/en/us/companion-moc.aspx> Site: searchable, easy-to-browse digital content with integrated premium online resources that 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 current premium content on TechNet, MSDN®, or Microsoft® Press®.



Additional Reading: Student Course files on the <http://www.microsoft.com/learning/en/us/companion-moc.aspx> Site:

http://www.microsoft.com/learning/en/us/companion-moc.aspx Site: includes the Allfiles.exe, a self-extracting executable file that contains all required files 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 an email to support@microsoftcourseware.com. To inquire about the Microsoft Certification Program, send an email to mchelp@microsoft.com.

Virtual Machine Environment

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

Virtual Machine Configuration

In this course, you will use Microsoft® Hyper-V™ to perform the labs.

 **Note:** At the end of each lab, you must revert the virtual machines to a snapshot. You can find the instructions for this procedure at the end of each lab

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

Virtual machine	Role
20761B-MIA-DCA	Domain controller for the ADVENTUREWORKS domain.
20761B-MIA-SQL	SQL Server and SharePoint Server

Software Configuration

The following software is installed:

- Microsoft Windows Server 2012 R2
- Microsoft SQL Server 2016
- Microsoft Office 2016
- Microsoft SharePoint Server 2013
- Microsoft Visual Studio 2015
- Microsoft Visio 2013

Course Files

The files associated with the labs in this course are located in the D:\Labfiles folder on the 20761B-MIA-SQL virtual machine.

Classroom Setup

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

Course Hardware Level

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

Hardware Level 6+

- Intel Virtualization Technology (Intel VT) or AMD Virtualization (AMD-V) processor
- Dual 120 GB hard disks 7200 RM SATA or better*
- RAM: 12GB or higher. 16 GB or more is recommended for this course.
- DVD drive

- Network adapter with Internet connectivity
- Super VGA (SVGA) 17-inch monitor
- Microsoft Mouse or compatible pointing device
- Sound card with amplified speakers

*Striped

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

MCT USE ONLY. STUDENT USE PROHIBITED

Module 1

Introduction to Microsoft SQL Server 2016

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	1-9
Lab: Working with SQL Server 2016 Tools	1-16
Module Review and Takeaways	1-19

Module Overview

This module provides an overview of Microsoft® SQL Server®, the data management software that stores data securely. Before you start, it is helpful to understand the basic architecture of SQL Server 2016, the different editions that are available, and a little about SQL Server Management Studio (SSMS). SSMS is one of the tools you use to connect to instances of SQL Server, write queries, and view data returned by your queries.

Objectives

After completing this module, you will be able to:

- Describe the architecture of SQL Server 2016.
- Describe the different editions of SQL Server 2016.
- Work with SSMS.

Lesson 1

The Basic Architecture of SQL Server

This lesson explains the basic architecture of Microsoft SQL Server, together with some key concepts. You will learn about SQL Server instances, the services, and how databases are structured. This will help you understand how SQL Server works before you start writing SQL Server queries.

Lesson Objectives

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

- Relational databases in general, and specifically the role and structure of SQL Server databases.
- The sample database used in this course.
- What is meant by the client server model.
- The structure of Transact-SQL (T-SQL) queries.

Relational Databases

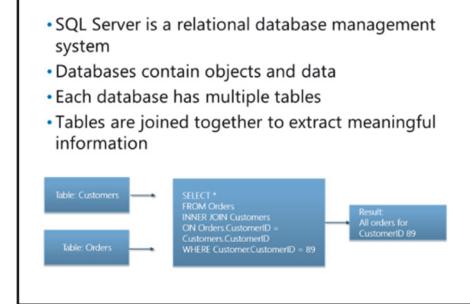
SQL Server 2016 is a data management system that uses the relational model to store and manage data. Relational databases store information in tables—each table holds information about just one thing. The information may concern something tangible, such as customer details, or intangible things such as orders.

You could hold customer information in the Customers table, but information about the goods they order would be in a separate table called Orders. This way of organizing data is efficient and removes redundant information. However, someone might ask to see all the orders placed by a particular customer. You use SQL Server to get this information by relating these tables to one another. You can then join the two tables together in a query to produce a list of all orders placed by a particular customer.

Databases typically have many different tables related to one another, so you often need to join several tables to obtain the information. For example, you might want to see the orders for customers who buy from one of your salespeople. You can do this by joining the Customers table, the Salesperson table, and the Orders table.

In addition to the databases that are created to store information, SQL Server includes five system databases:

- **master**: the system configuration database.
- **model**: the template database. SQL Server will apply any changes made in model to new databases.
- **msdb**: used by SQL Server Agent to schedule jobs and alerts.
- **tempDb**: a temporary store for data such as work tables. This database is dropped and recreated each time SQL Server restarts, which means that any temporary tables will be lost when SQL Server closes down.
- **resource**: a hidden, read-only database that contains all the system objects for other databases.

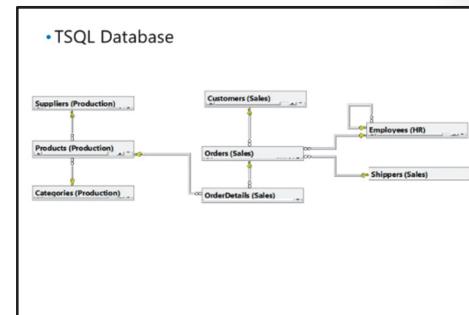


SQL Server databases contain data and objects, including tables, views, stored procedures, user accounts, and other management objects. Before you can execute queries, or insert or delete information from a database, you must connect to the database. You need security credentials to log on to SQL Server, and a database account with permissions to access data objects.

About the Course Sample Database

To understand how queries work, you will be using a database called TSQL. This is a small database suitable for learning how to write Transact-SQL queries. TSQL contains several types of objects:

- **Schemas.** These are logical containers for tables and views.
- **Tables.** These mostly relate to one another using Foreign Key constraints.
- **Views.** These display information from more than one table.



The TSQL database is a simple sales application for a small business. Some of the tables you will be working with include:

- **Sales.Orders.** This table stores invoice header information, such as a unique reference for the order, the customer who placed the order, and the date of the order.
- **Sales.OrderDetails.** This table stores transaction details about each order, such as products ordered, and the price.
- **Sales.Customers.** This table stores information about customers, such as company name, and contact details.
- **HR.Employees.** This table stores employee information.

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 the client application sends requests to the database engine as T-SQL statements, SQL Server performs the necessary file access, memory management, and processor utilization on behalf of the client. The client never has direct access to database files—unlike, for example, a desktop database application.

- The client software is separate from the server database engine
- Client/Server refers to the separation of functionality—not where the software is actually located
- Client software and server database engine can be on the same machine
- Databases can access data in other databases over a network

In this course, the client and server software are running on the same virtual machine but, in production environments, the client software runs on a separate machine to the database engine. Indeed, there could be multiple clients accessing the same server database engine.

Wherever the client and server software is located, it makes no difference to the way you write T-SQL code. On the logon screen, you just specify the SQL Server that you want to connect to.

You can also refer to other databases in a T-SQL script by using its four-part name. A four-part name has the format **Instance.Database.Schema.Object**. For example, the four-part name **MIA-SQL.sales dbo.orders** refers to the orders table, in the dbo schema, in the sales database, on the MIA-SQL server's default instance.

Connecting to a remote server requires the remote instance to be set up as a linked server. In T-SQL, you add a linked server using **sp_addlinkedserver**. Although **sp_addlinkedserver** takes a number of optional arguments, in its simplest form 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, which means it does not extract data row by row, but instead extracts data from tables that normally contain many rows. Only after it has retrieved the table does SQL Server filter data to produce a subset of the table, if that is what the query has requested. This makes SQL Server highly efficient in dealing with large volumes of data, but it means you have to think in sets to write efficient T-SQL code.

T-SQL scripts are stored in script files with a **.sql** extension. Inside each file, you can divide the script into batches, each batch concluding with the **GO** keyword. SQL Server runs each batch in its entirety before it starts the next one. This is important if you are relying on things happening in a specific order. For example, you must create a table before you can populate the table with data. To complete these two steps within the same script file, you must specify the table structure first, and then add data to the table. If you try to create the table and populate it with data without the **GO** keyword in between, the statement will fail. It will succeed only when the **GO** keyword completes the first **CREATE TABLE** statement before the **INSERT INTO** statement populates the table with data.

- T-SQL is a set-based language
- T-SQL is written in scripts with **.sql** extension
- **GO** keyword separates batches

```
CREATE TABLE dbo.Employees
(
    EmployeeID int PRIMARY KEY,
    LastName nvarchar(25),
    FirstName nvarchar(25)
)
GO

INSERT INTO dbo.Employees
(EmployeeID, LastName, FirstName)
VALUES
(121, N'O'Neil', N'Carlene')
GO
```

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
	CREATE TABLE HR.Employees (EmployeeID int PRIMARY KEY, LastName nvarchar(25), FirstName nvarchar(25));
	GO
	INSERT INTO HR.Employees (EmployeeID, LastName, FirstName) VALUES (121, N'O'Neill, N'Carlene');
	GO

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 about the different editions of SQL Server 2016 that are available, their distinguishing features, and which edition might be best 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 2016.

SQL Server Versions

SQL Server 2016 is the latest version in SQL Server's development. Since it was first developed in 1989 for the OS/2 operating system, SQL Server has gone through a number of major releases. SQL Server 4.2 and later versions were developed to run on Windows®.

The SQL Server Database Engine had major enhancements for version 7.0 and all subsequent versions have continued to extend and improve SQL Server functionality, making it suitable for workgroup and enterprise use.

SQL Server 2016 is a major new release with enhanced security, support for hybrid cloud installations, and major improvements in the analytics functionality.

SQL Server Versions	
Version	Release Year
2016	2016
2014	2014
2008 R2	2010
2008	2008
2005	2005
2000	2000
7.0	1998
6.5	1996
6.0	1995
4.2.1	1994
4.2	1992
1.1	1991
1.0	1989



Note: Although the name may suggest otherwise, SQL Server 2008 R2 is not a service pack for SQL Server 2008. SQL Server 2008 R2 is an independent version (number 10.5) with enhanced multiserver management capabilities, in addition to new business intelligence (BI) features.

Question: Which version of SQL Server are you currently working with? Have you worked with any earlier versions?

SQL Server Editions

SQL Server is available in different editions with different feature sets that target various business scenarios. In the SQL Server 2012 release, Microsoft streamlined the number of editions from previous versions. Four main editions are available:

- **Enterprise.** This is SQL Server's flagship edition containing all features, including Business Intelligence, support for data warehousing, and high availability.
- **Standard.** This includes the database engine, as well as core reporting and analytics capabilities. Standard supports 16 processor cores but does not include all the high availability, security, and data warehousing features found in the Enterprise edition.
- **Business Intelligence.** This includes the core database engine, along with the full Business Intelligence functionality of analytics, reporting, and integration services. However, like the Standard edition, it supports 16 processor cores and does not offer all the high availability, security, and data warehousing features of the Enterprise edition.
- **Express.** This is a free version of SQL Server and is limited to four processor cores, 1 GB of memory per instance, and 10 GB maximum storage per database.

This course uses features that are found in all editions.

In addition to the editions described above, SQL Server also runs in the cloud, in one of two ways:

- You can install SQL Server on a cloud-based virtual machine that your organization has provisioned and integrated with its infrastructure. When properly set up, SQL Server works as if it were on a server on your network.
- Secondly, it could be an Azure SQL Database. This is Software as a Service (SaaS) and allows you to use SQL Server without a physical server or a cloud-based virtual machine. You can add or remove performance as required, making this a highly scalable option.

 **Additional Reading:** For more information on the use of T-SQL on Microsoft Azure SQL Server Database, see the MSDN article *Azure SQL Database Transact-SQL Information* at: <https://azure.microsoft.com/en-gb/documentation/articles/sql-database-transact-sql-information/>

 **Microsoft Azure SQL Server Database**

<http://aka.ms/ybpqh8>

• SQL Server Editions

Main Editions	Other Editions
Enterprise	Developer
Standard	Express
Business Intelligence	Compact
Azure SQL Database	

MCT USE ONLY. STUDENT USE PROHIBITED

Check Your Knowledge**Question**

You have founded a new company with two friends. Your new application (app) uses a SQL Server database to store information. You are unsure whether your app will be successful but, if it is, you will need both high performance and space for large volumes of data. However, you have not yet launched, so are unsure how many people will use your app. Which edition of SQL Server 2016 should you use for this system?

Select the correct answer.

	Azure SQL Database
	Enterprise edition
	Express edition
	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. You will 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 SQL Server 2016 Technical Documentation.

Starting SSMS

SSMS is an integrated management, development, and querying application that has many features for exploring and working with your databases. Microsoft based SSMS on the Visual Studio shell; if you know Visual Studio, you will most likely feel comfortable using SSMS.

You can start SSMS in one of two ways:

- Use the **SSMS** shortcut on the Windows Start menu.
- Type **ssms.exe** in a command prompt window.

- Launch SSMS from the Windows Start screen
 - Or type **SSMS** into the Search Programs and Files box
- Connect to a SQL Server instance
 - Or work disconnected
- Settings available in Tools, Options include:
 - Fonts and colors, line numbering, and word wrap
 - Which windows open when SSMS is launched
- Useful windows include:
 - Query Editor
 - Object Explorer
 - Solution Explorer

By default, SSMS will display a Connect to Server dialog box where you can specify the server (or instance) name, together with your security credentials. To specify the database you want to connect to, click the **Options** button to open the Connection properties dialog box. Alternatively, you can select the database after you have connected.

You can explore many SSMS features without connecting to a SQL Server instance. You can also cancel the Connect to Server dialog box if you want to connect to a server later.

With SSMS running, you can explore some of its settings found in Tools, Options. You can change the default font, enable line numbering for scripts, and control the behavior of its many windows.

For more information on using SSMS, see *Use SQL Server Management Studio* in SQL Server 2016 Technical Documentation.



Use SQL Server Management Studio

<http://aka.ms/cbalxi>

Connecting to SQL Server

To connect to an instance of SQL Server, you need to specify several items, regardless of how you connect:

- The instance you want to connect to. This must be in the format: **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 a database, you will connect to the default database designated by the database administrator. If no default is assigned, you will connect to the master database.

Question: In your organization, which authentication method do you use to log on to SQL Server?

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

Working with Object Explorer

Object Explorer is a graphical tool for managing SQL Server instances and databases. It is one of several SSMS windows available from the View menu. Object Explorer provides direct interaction with most SQL Server data objects such as tables, views, and stored procedures. To display context-sensitive help for an object in the tree view, right-click an object, such as a table. The available options include query and script generators for object definitions.

- Object Explorer is a hierarchical, graphical view of SQL Server objects
- Explore objects in the default instance, and additional named instances
- Right-click for context-sensitive menu with frequently used commands
- Create T-SQL scripts of object definitions, and send to the query window, clipboard or a file
- Start a new query window by right-clicking a database
- Changing the selected object does not change the existing connection



Note: Operations performed in SSMS require 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.

Use Object Explorer to learn about the structure and definition of data objects you want to use in your queries. For example, to see the names of the columns in a table:

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

5. Locate the relevant table and expand it to find the Columns folder.
6. The Columns folder displays the names, data types, constraints, and other information about the column definition.
7. To avoid typing, drag an object from the Object Explorer hierarchy into the query window.



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

Script Files and Projects

SSMS allows you to create and save T-SQL code in text files with a **.sql** file extension. Like other Windows applications that open, edit, and save files, SSMS has both a File menu and toolbar buttons.

In addition to working on individual script files, SSMS lets you organize files into solutions and projects. You can keep scripts for one project together, saving time by opening or closing all the files at the same time. You can open solutions, projects, or script files from SSMS or File Explorer.

- T-SQL scripts are text files with a **.sql** extension
- SSMS can open, edit, and execute code in script files
- SSMS allows you to organize script files into:
 - Solutions (*.ssmssqln)
 - Projects (*.ssmssqlproj)
- Opening a solution is a convenient way to open all relevant files
- You will use projects on this course

Object	Parent	Description
Solution	-	A solution is a conceptual container for projects. Solutions have a .ssmssqln extension, and are always displayed at the top of the hierarchy.
Project	Solution	Projects contain queries (T-SQL scripts), database connection metadata, and other miscellaneous files. You can file any number of projects within a solution. Projects have a .ssmssqlproj extension.
Script	Project	T-SQL script files with a .sql extension are the basic files used to work with SQL Server.

To create a new solution, click the **File** menu and click **New Project**. There is no “New Solution” command; if you want a new solution, select **Create New Solution** in the **New Project** dialog box. Type the name for the project, the parent solution, and whether you want the project to be stored in a subfolder within the solution. Click **OK** to create the objects.

You can view solutions and projects by opening the View menu, and selecting Solution Explorer. Solution Explorer displays a hierarchical list of all the solutions and projects you have created.

To create a new script that will be stored as part of a project, right-click the **Queries** folder in the **Project** and select **New Query**.



Note: When you create a new query using the toolbar button or the New Query command on the File menu, the script file is stored in the Miscellaneous Files folder by default. Use the **Save As** menu option to save the file in your preferred location. You can drag files from the Miscellaneous Files folder to specific projects, using Solution Explorer to put a copy of the file into a specific project folder. Alternatively, you can move the file by holding the Alt key as you drag.

Remember to save the solution when exiting SSMS, or when opening another solution. When you save a script using the Save toolbar button or the Save <queryname>.sql command on the File menu, you are only saving changes to the current script file. To save the solution and its contents, use the Save All command on the File menu or, when prompted, save the **.ssmssqln** and **.ssmssqlproj** files on exit.

Executing Queries

To execute T-SQL code in SSMS, open the .sql file that contains the query, or type your query into a new query window. You can either run all of the script or part of it:

- Select the portion of code you wish to execute.
- If you do not select anything, the entire script will run.

When you have decided what you wish to execute, run the code by either:

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

By default, SSMS will display the results in a new pane of the query window. To change the location and appearance of the results, click **Tools**, and then click **Options**. CTRL-R toggles between a full screen T-SQL editor, and the T-SQL editor plus the results pane.

SSMS enables results to be displayed in three different ways:

- **Grid.** A spreadsheet-like view, with row numbers and columns you can resize. Use **Ctrl+D** to select Grid layout before executing the query.
- **Text.** A Windows Notepad-like display that pads column widths. Use **Ctrl+T** to select text layout before executing the query.
- **File.** Saves query results to a text file with a **.rpt** extension. When you execute the query, you will be prompted for a location to save the file. You can then open the file with any application that reads text files, such as Windows Notepad and SSMS. To send results to file, use the keyboard shortcut **Ctrl-Shift-F** before running the query.

- To execute queries in SSMS:
 - Open a saved script, or write a new query
 - Three ways to execute the query:
 - From the Query menu, select **Execute**
 - Press **F5**
 - Click the **Execute** toolbar button





Note: The shortcut to display results as text has changed in SQL Server 2016. It is now **Ctrl+T**. It used to be Ctrl+F.



Additional Reading: For a list of keyboard shortcuts available in SSMS, see *SQL Server Management Studio Keyboard Shortcuts*, in MSDN.



MSDN SQL Server Management Studio Keyboard Shortcuts

<http://aka.ms/y83i8i>

Using SQL Server 2016 Technical Documentation

SQL Server 2016 Technical Documentation (sometimes abbreviated to BOL) is the product documentation for SQL Server. BOL includes helpful information on SQL Server's architecture and concepts, in addition to syntax reference for T-SQL. You can start BOL from the Help menu in SSMS, or from the query window. For context-sensitive help for T-SQL keywords, select the keyword and press F1.

- Books Online (BOL) is the product documentation for SQL Server
- In SQL Server 2016, Books Online does not ship with the SQL Server installation media
- Configure Help to display content from MSDN or download Help Collections to a local computer
- Once configured, Help is available from:
 - SSMS query window (context-sensitive when you highlight a keyword)
 - SSMS Help menu
 - Windows Start menu

You can also view SQL Server 2016 Technical Documentation on the Microsoft website.



SQL Server 2016 Technical Documentation

<http://aka.ms/dxlgb>



Note: Before SQL Server 2014, there was a setup option to install SQL Server Books Online locally. In SQL Server 2016, you must download and install SQL Server 2016 Technical Documentation separately.

The first time you invoke Help, you will be prompted to specify whether you want to view SQL Server 2016 Technical Documentation content locally or online. When you work with online help, you will always access the latest information.

For detailed instructions on how to download, install, and configure SQL Server 2016 Technical Documentation for local offline use, see *Get Started with Product Documentation for SQL Server*.



Get Started with Product Documentation for SQL Server

<http://aka.ms/tgv2o6>

Demonstration: Introducing Microsoft SQL Server 2016

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 2016

1. Ensure that the **20761B-MIA-DC** and **20761B-MIA-SQL** virtual machines are running.
2. Log on to **20761B-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
3. In the **D:\Demofiles\Mod01** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
4. In the **User Account Control** dialog box, click **Yes**, press **y** when prompted, and then press **Enter**.
5. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.

Explore Database and Other Objects

1. In Object Explorer, expand the **Databases** folder to see a list of databases.
2. Expand the **TSQL** database.
3. Expand the **Tables** folder.
4. Expand the **Sales.Customers** table.
5. Expand the **Columns** folders.
6. View the list of columns, and the data type information for each column.
7. Note the data type for the **companynam** column.

Work with T-SQL Scripts

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

```
USE TSQL;
GO
SELECT CustID, ShipCountry
FROM Sales.Orders;
```

11. Select the code and click **Execute**.

12. Point out the **Results** pane.
13. On the **File** menu, click **Save All**.
14. On the **File** menu, click **Close Solution**.
15. On the **File** menu, point to **Recent Projects and Solutions**, and then click **1 D:\...\Module 1 Demonstration\Module 1 Demonstration.ssmssln**.
16. Point out the **Solution Explorer** pane.
17. Close SQL Server Management Studio without saving any files.

Check Your Knowledge

Question	
A colleague has asked you to run some test queries against the company's scheduling database. Administrators have given you the name of the server where the database is hosted, and the name of the database. Permissions to run the necessary queries have been granted to your Active Directory® account. You are logged on to 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.	
	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.

Lab: Working with SQL Server 2016 Tools

Scenario

The Adventure Works Cycles Bicycle Manufacturing Company has adopted SQL Server 2016 as its relational database management system. You need to retrieve business data from several SQL Server databases. In the 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 2016 Technical Documentation.

Estimated Time: 30 minutes

Virtual machine: **20761B-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 want to become familiar with the development environment. You have therefore 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. With SSMS running, open the **Tools** menu and choose **Options**. Change the text editor font size to **14** points.
2. Change additional settings in **Options**:
 - Disable IntelliSense.
 - Change the tab indent to 6 spaces.
 - Include column headers when copying the result set from the grid. Select **Query Results, SQL Server, Results to Grid**. Select **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

You have decided to organize your T-SQL scripts in a project folder. In this lab, you will practice how to create a project and add 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 save it in the folder **D:\Labfiles\Lab01\Starter**.
2. Add a new query called **MyFirstQueryFile.sql** to **MyFirstProject**.
3. Save the project and the query file by clicking **Save All**.

► Task 2: Add an Additional Query File

1. Add an additional query file called **MySecondQueryFile.sql** to the project you created.
2. Open File Explorer, navigate to the **MyFirstProject** folder to check that your second query file is in your project folder.
3. In SSMS, use the Solution Explorer pane to remove **MySecondQueryFile.sql** from your project by choosing the **Remove** option. (Not the Delete option.)
4. In File Explorer, check to see whether the second query file is still in the project folder.
5. In SSMS, remove **MyFirstQueryFile.sql** by choosing **Delete**.
6. To see the difference, check in File Explorer.

► Task 3: Reopen the Created Project

1. Save the project, close SSMS, reopen SSMS, and open the project **MyFirstProject**.
2. Drag **MySecondQueryFile.sql** from File Explorer to the Queries folder beneath **MyFirstProject** in SSMS Solution Explorer.
3. Save the project.

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

Exercise 3: Using SQL Server 2016 Technical Documentation

Scenario

To be effective in your upcoming training and exercises, you will practice using SQL Server 2016 Technical Documentation to check T-SQL syntax.

The main tasks for this exercise are as follows:

1. Launch SQL Server 2016 Technical Documentation
2. Use SQL Server 2016 Technical Documentation

► **Task 1: Launch SQL Server 2016 Technical Documentation**

1. Launch Manage Help Settings from the Windows Start screen.
2. Configure SQL Server 2016 Technical Documentation to use the online option, not local help.

► **Task 2: Use SQL Server 2016 Technical Documentation**

- Use SQL Server 2016 Technical Documentation to find information about what's new in the SQL Server 2016 Database Engine.

Results: After this exercise, you will understand how to find the information you need in SQL Server 2016 Technical Documentation.

Module Review and Takeaways

In this module, you have learned how to:

- Describe the architecture of SQL Server 2016.
- Describe the different editions of SQL Server 2016.
- Work with SSMS.

Review Question(s)

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

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

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

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	2-17
Lesson 4: Understanding the Logical Order of Operations in SELECT Statements	2-19
Lab: Introduction to T-SQL Querying	2-24
Module Review and Takeaways	2-27

Module Overview

Transact-SQL, or T-SQL, is the language you will use to interact with Microsoft® SQL Server® 2016. In this module, you will learn that T-SQL has many elements in common with other computer languages, such as commands, variables, loops, functions, and operators. You will also learn that designing your queries to take sets into account means SQL Server will perform at its best. To make the most of your effort in writing T-SQL, you will also learn the process and order by which SQL Server evaluates your queries. Understanding the logical order for operations of SELECT statements is 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 also learn about many of the T-SQL language elements and which ones are 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. The language was originally developed to support the new relational data model at International Business Machines (IBM) in the early 1970s. Since then, 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 the 1970s
 - Adopted by ANSI and ISO standards bodies
 - Widely used in the industry
 - PL/SQL (Oracle), SQL Procedural Language (IBM), Transact-SQL (Microsoft)
- Transact-SQL is commonly referred to as T-SQL
 - The querying language of SQL Server 2016
 - SQL is declarative
 - Describe what you want, not the individual steps

The ANSI standard has gone through several revisions, including SQL-89 and SQL-92—the specifications are either fully or partly supported by T-SQL. SQL Server 2016 also implements features from later standards, such as ANSI SQL-2008 and ANSI SQL-2011. 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 not a procedural language but a set-based and declarative language. 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 supplying 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 approach might look like this:

Procedural Approach

```
Loop through each row in the data.  
If the city is Portland, return the row; otherwise do nothing.  
Move to next row.  
End loop.
```

This procedural code has to contain the logic to retrieve the data—to inspect the data to see if it meets your needs—and will be doing this for all the data in the table, whether or not it is relevant.

With a declarative language such as T-SQL, you will provide the attributes and values that describe the set you wish to retrieve. You do not have to specify how to retrieve the data, but you should identify what the data is.

For example, see the following pseudo-code:

Declarative Language

Return all the customers whose city is Portland

With T-SQL, the SQL Server 2016 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.

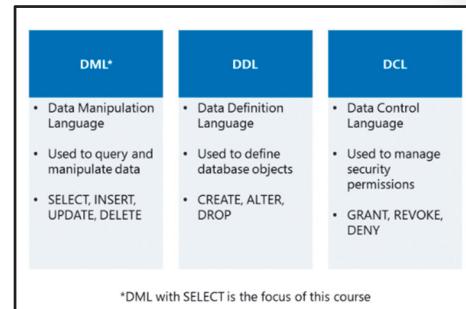
If you have a background in other programming environments, adopting a new mindset may present a challenge. This course has been designed to help you bridge the gap between procedural and set-based, declarative T-SQL.

 **Note:** Sets will be discussed later in this module.

Categories of T-SQL Statements

T-SQL statements can be organized into three categories:

- **Data Manipulation Language (DML)** is the set of T-SQL statements that focuses on querying and modifying data. This includes SELECT, the primary focus of this course, and modification statements such as INSERT, UPDATE, and DELETE. You will learn about SELECT statements throughout this course.
- **Data Definition Language (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 (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 2016 security, including DCL, see the Microsoft Official Course 20473-2: *Administering a SQL Database Infrastructure*.

For additional information on DML, DDL, and DCL commands, see the SQL Server 2016 Technical Documentation:

 **Transact-SQL Reference (Database Engine)**

<http://aka.ms/hjmhuj>

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 might 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.



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

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

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.

Predicates supported by T-SQL include the following:

- **IN:** used to determine whether a value matches any value in a list or subquery. For example, WHERE day IN (1,5,6,10).
- **BETWEEN:** used to specify a range of values. For example, WHERE rate BETWEEN 3 AND 7.
- **LIKE:** used to match characters against a pattern. For example, WHERE surname LIKE '%mith%'.

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.)
- **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: =.

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

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	Operator
1	\sim (Bitwise NOT)
2	$/$, $*$, $\%$ (Division, Multiply, Modulo)
3	$+$, $-$, $\&$, \wedge , $ $ (Positive/Add/Concatenate, Negative/Subtract, Bitwise AND, Bitwise Exclusive OR, Bitwise OR)
4	$=$, $>$, $<$, $<=$, $<$, $!=$, $!$, $!<$, $!>$ (Comparisons)
5	NOT
6	AND
7	ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
8	$=$ (Assignment)

For more information on operator precedence, see the SQL Server 2016 Technical Documentation:



Operator Precedence

<http://aka.ms/y6ylxo>

T-SQL Language Elements: Functions

SQL Server 2016 provides a wide variety of functions for 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
 - STUFF
 - SOUNDEX

String Functions	Date and Time Functions	Aggregate Functions
<ul style="list-style-type: none"> • SUBSTRING • LEFT, RIGHT • LEN • REPLACE • REPLICATE • UPPER, LOWER • LTRIM, RTRIM • STUFF • SOUNDEX 	<ul style="list-style-type: none"> • GETDATE • SYSDATETIME • GETUTCDATE • DATEADD • DATEDIFF • YEAR • MONTH • DAY • DATENAME • DATEPART • ISDATE 	<ul style="list-style-type: none"> • SUM • MIN • MAX • AVG • COUNT • COUNT_BIG • STDEV • STDEVP • VAR

MCT USE ONLY STUDENT USE PROHIBITED

- Date and time functions
 - GETDATE, SYSDATETIME, GETUTCDATE
 - DATEADD, DATEDIFF
 - YEAR, MONTH, DAY
 - DATENAME, DATEPART
 - ISDATE
- Aggregate functions
 - SUM, MIN, MAX, AVG
 - COUNT, COUNT_BIG
 - STDEV, STDEVP
 - VAR
- 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 is presented here at a high conceptual level. Subsequent modules in this course will show more detailed explanations.

For more information, including code samples, see the SQL Server 2016 Technical Documentation:



Built-in Functions

<http://aka.ms/jw8w5j>

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 give 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 2016, you can declare and initialize a variable in the same statement

```
DECLARE @search varchar(30) = 'Match%';
```



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, because 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.

The following example creates a character variable initialized to the string 'Program%':

Character Variable

```
DECLARE @search varchar(30) = 'Match%';
```

The following example creates a date variable and assigns the current date:

Date Variable

```
DECLARE @CurrentDate date;
SET @CurrentDate = GETDATE();
```

You will learn more about different data types—including dates—and 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 temporary and permanent storage in databases.

For more information on different types of tables, see:



Types of Tables in SQL 2016

<http://aka.ms/quew7f>

T-SQL Language Elements: Expressions

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

- 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.

- 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 the same data type

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



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.

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;
```

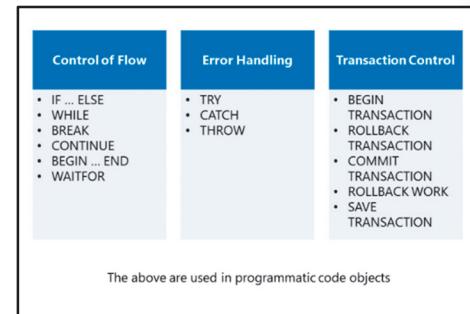
 **Note:** The preceding example uses T-SQL techniques, such as column aliases and date functions, which will be covered later in this course.

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 the structure of exception handling (error handling).
- **THROW**, for raising an exception and transferring execution to a CATCH block.
- **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. For example, if your primary role is as a report writer, 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 use these elements.

T-SQL Language Elements: Comments

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

- Two methods for marking text as comments
 - A block comment, surround text with /* and */

```
/* All the text in this paragraph will be treated as
   comments by SQL Server.
*/
```
 - An inline comment, precede text with --

```
-- This is an inline comment
```

• Many T-SQL editors will color comments as above

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

Example of a single line comment:

Single Line Comments

```
-- This whole line is a comment
DECLARE @search varchar(30) = 'Match%'; -- end of a line
```

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

Example of a block comment:

Block Comment

```
/*
   All the text in this paragraph will be treated as comments
   by SQL Server.
*/
```

Many query editing tools, such as SQL Server Management Studio (SSMS), Visual Studio®, or SQLCMD, will color-code commented text in a different color from 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.

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, because you will be submitting a single query at a time. However, when you need to create and manipulate objects, you might need to separate statements into distinct batches. For example, a CREATE VIEW statement might 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 table1 (col1 int);
CREATE VIEW view1 as SELECT * FROM table1;
```

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. Insert a GO batch separator between the two CREATE statements to resolve the previous error.

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

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 5
Must declare the scalar variable "@custid".
```

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. In File Explorer, browse to **D:\Demofiles\Mod02**, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**.
4. When the script has finished, press any key.
5. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows® Authentication.
6. On the **File** menu, point to **Open**, and then click **Project/Solution**.
7. In the **Open Project** dialog box, browse to the **D:\Demofiles\Mod02\Demo** folder, and then double-click **Demo.ssmssln**.
8. In Solution Explorer, expand **Queries**, and then double-click the **11 - Demonstration A.sql** script file.
9. Select the code under **Step 1**, and then click **Execute**.
10. Select the code under **Step 2**, and then click **Execute**.
11. Select the code under **Step 3**, and then click **Execute**.
12. Select the code under **Step 4**, and then click **Execute**.
13. Select the code under **Step 5**, and then click **Execute**.
14. Select the code under **Step 6**, and then click **Execute**.

15. Select the code under **Step 7**, and then click **Execute**.
16. Select the code under **Step 8**, and then click **Execute**.
17. Select the code under the comment **Cleanup task if needed**, and then click **Execute**.
18. Close SQL Server Management Studio.

Check Your Knowledge

Question	
From the following T-SQL elements, select the one that does not contain an expression:	
Select the correct answer.	
	SELECT FirstName, LastName, SkillName AS Skill, GetDate() - DOB AS Age
	WHERE HumanResources.Department.ModifiedDate > (SYSDATETIME() - 31)
	JOIN HumanResources.Skills ON Employees.ID = Skills.EmployeeID
	WHERE Skill.Level + Skill.Confidence > 10

Lesson 2

Understanding Sets

This lesson introduces the concepts of the set theory, one of the mathematical underpinnings of relational databases, and helps 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.

The Set Theory and SQL Server

The set theory is one of the mathematical foundations of the relational model and so is fundamental to working with SQL Server 2016. 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.

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.

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

As you learn more about certain T-SQL query statements, it is 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. Depending on your background, this may require an adjustment.

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, when you start working with subsets of data, it's important to know how you can uniquely address each member of a set.

Characteristics of a Set	Example
Elements of a set called Members	Customer as a member of set called Customers
Elements of a set are described by attributes	First name, Last name, Age
Elements must be unique	Customer ID
Set theory does not specify the order of its members	

NCT 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, when you write T-SQL code, you are addressing a collection of elements, 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.

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:

- 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.
- When possible, ensure that you are addressing elements via their unique identifiers, such as keys. 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.

Application of Set Theory	Comments
Acts on all elements at once	Query the whole table
Use set-based processing	Tell the engine what you want to retrieve
Avoid cursors or loops	Do not process each item individually
Members of a set must be unique	Define unique keys in a table
No defined order to result set	Use ORDER BY clause if results need to be ordered



Additional Reading: For more information on set theory and logical query processing, and its application to SQL queries, see Chapter 1 of Itzik Ben-Gan's *T-SQL Querying* (Microsoft Press, 2015).

MCT USE ONLY STUDENT USE PROHIBITED

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

Lesson 3

Understanding Predicate Logic

Along with set theory, predicate logic is a mathematical foundation for the relational database model, and with it, SQL Server 2016. You probably have a fair amount of experience with predicate logic—rather than the set theory—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

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 need to account for missing or unknown values, and 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 discussed further in the next topic, and later in this course.

- Predicate logic is another 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

MCT USE ONLY. STUDENT USE PROHIBITED

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 they have been discussing them separately, predicates do not stand alone, syntactically speaking. Typically, you will 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, you 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; the interaction of T-SQL predicates with missing values results in an unknown. When you are designing query logic, ensure that you have accounted for all three possible outcomes—true, false, or unknown. You will learn how to use three-valued logic in WHERE clauses later in this course.

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
	WHILE statements
	All of the above

Uses for Predicates

- Filtering data in queries
- Providing conditional logic to CASE expressions
- Joining tables
- Defining subqueries
- Enforcing data integrity
- Control of flow

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, providing 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 understand the logical order of operations, you 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 to recognize its place in the sequence of events.

A SELECT statement is made up of mandatory and optional elements. Strictly speaking, SQL Server only requires a SELECT clause to execute without error. A SELECT clause without a FROM clause operates as if selecting from an imaginary table containing one row. You will see this behavior when you test variables later in this course. However, as a SELECT clause without a FROM clause cannot retrieve data from a table, you will treat stand-alone SELECT clauses as a special case 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 others present. For example, a WHERE clause will always be evaluated after the FROM clause and before a GROUP BY clause, if one exists.

You will discuss the order of these operations in the next topic.

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



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 it is evaluated and processed by the SQL Server Database Engine.

Consider the following query:

Logical Query Processing

```
USE TSQl;
SELECT EmployeeId, YEAR(OrderDate) AS
OrderYear
FROM Sales.Orders
WHERE CustomerId = 71
GROUP BY EmployeeId, YEAR(OrderDate)
HAVING COUNT(*) > 1
ORDER BY EmployeeId, OrderYear;
```

5.	SELECT	<select list>
1.	FROM	<table source>
2.	WHERE	<search condition>
3.	GROUP BY	<group by list>
4.	HAVING	<search condition>
6.	ORDER BY	<order by list>

The order in which a query is written is not the order in which it is evaluated by SQL Server

Before you 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 you're connected to the correct database for the query. This line is not being examined for its run-time order.

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

Change the Database Connection

```
USE TSQl; -- change connection context to a database named TSQl.
```

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

The SELECT clause returns the EmployeeId column and extracts just the year from the OrderDate column:

Start of SELECT

```
SELECT EmployeeId, YEAR(OrderDate) AS OrderYear
```

The FROM clause identifies which table is the source of the rows for the query—in this case Sales.Orders:

FROM Clause

```
FROM Sales.Orders
```

The WHERE clause filters the rows out of the Sales.Orders table, keeping only those that satisfy the predicate—in this case, a customer with an ID of 71:

WHERE Clause

```
WHERE CustomerId = 71
```

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

GROUP BY Clause

```
GROUP BY EmployeeId, 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
```

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

ORDER BY Clause

```
ORDER BY EmployeeId, OrderYear;
```

Now that you'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, you'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 (in our example, OrderYear) created there cannot be used in the GROUP BY or HAVING clause.

6. 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

```
FROM Sales.Orders
WHERE CustomerId = 71
GROUP BY EmployeeId, YEAR(OrderDate)
HAVING COUNT(*) > 1
SELECT EmployeeId, YEAR(OrderDate) AS OrderYear
ORDER BY EmployeeId, OrderYear;
```

NO TUTORIALS

As you have seen, you do not write T-SQL queries in the same order in which they are logically evaluated. Because the run-time 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 queries.

Applying the Logical Order of Operations to Writing SELECT Statements

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

- Decide which tables to query first, in addition to 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. Remember that only columns in the GROUP BY clause, in addition to 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—you will need to add an ORDER BY clause to guarantee a sort order if required.

```
USE TSQl;
SELECT EmployeeId, YEAR(OrderDate) AS OrderYear
FROM Sales.Orders
WHERE CustomerId = 71
GROUP BY EmployeeId, YEAR(OrderDate)
HAVING COUNT(*) > 1
ORDER BY EmployeeId, OrderYear;
```

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. Start the 20761B-MIA-DC, 20761B-MIA-SQL, MSL-TMG1 virtual machines, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Open SQL Server Management Studio.
3. In the **Connect to Server** dialog box, in the **Server name** box, enter the server you created during preparation. For example, **20761Ba-azure.database.windows.net**.
4. In the **Authentication** list, click **SQL Server Authentication**.

5. In the **Login** box, type **Student**.
6. In the **Password** box, type **Pa\$\$w0rd**, and then click **Connect**.
7. On the **File** menu, point to **Open**, and then click **Project/Solution**.
8. In the **Open Project** dialog box, browse to the **D:\Demofiles\Mod02\Demo** folder, and then double-click **Demo.ssmssln**.
9. In Solution Explorer, double-click the **21 - Demonstration B.sql** script file.
10. On the **Query** menu, point to **Connection**, and then click **Change Connection**.
11. In the **Connect to Database Engine** dialog box, in the **Server name** box, enter the server you created during preparation. For example, **20761Ba-azure.database.windows.net**.
12. In the **Authentication** list, click **SQL Server Authentication**.
13. In the **Login** box, type **Student**.
14. In the **Password** box, type **Pa\$\$w0rd**, and then click **Connect**.
15. In the **Available Databases** list, click **AdventureWorksLT**.
16. Select the code under the comment **Step 1**, and then click **Execute**.
17. Select the code under **Step 2**, and then click **Execute**.
18. Select the code under **Step 3**, and then click **Execute**.
19. Select the code under **Step 4**, and then click **Execute**.
20. Select the code under **Step 5**, and then click **Execute**. Note the error message.
21. Select the code under **Step 6**, and then click **Execute**.
22. Select the code under **Step 7**, and then click **Execute**.
23. Select the code under **Step 8**, and then click **Execute**.
24. Close SQL Server Management Studio, without saving any changes.

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
	FROM
	WHERE
	GROUP BY
	HAVING
	SELECT
	ORDER BY

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Introduction to T-SQL Querying

Scenario

You are an Adventure Works business analyst, who will be writing reports against corporate databases stored in SQL Server 2016. To help you become more comfortable with SQL Server querying, the Adventure Works IT department has provided some 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: **20761B-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—this includes the firstname, lastname, city, and country columns. You will execute the T-SQL script against the TSQl 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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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. Open the project file **D:\Labfiles\Lab02\Starter\Project\Project.ssmssln**.
2. Connect to the **MIA-SQL** database using Windows authentication.
3. Open the T-SQL script **51 - Lab Exercise 1.sql**.
4. Execute the whole script.
5. Observe the result and the database context.
6. Which database is selected in the Available Databases box?

► **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**.



Note: 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.

3. Close all open windows.

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. Change the Database Context with the GUI
3. Change the Database Context with T-SQL

► **Task 1: Execute the T-SQL Script**

1. 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.
2. There is an error. What is the error message? Why do you think this happened?

► **Task 2: Change the Database Context with the GUI**

1. Apply the needed changes to the script so that it will run without an error. (Hint: you do not need to change any T-SQL information to fix the error.) 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: Change the Database Context with T-SQL**

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 following statements:

```
USE TSQl;
GO
```

3. Save and close the T-SQL script. Re-open the T-SQL script **61 - Lab Exercise 2.sql**. 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.



Note: SSMS supplies keyboard shortcuts and two buttons so you can quickly comment and uncomment code. The keyboard shortcuts are CTRL+K then CTRL+C to comment, and CTRL+K then CTRL+U to uncomment.



Or you can use these buttons on the toolbar.

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

► Task 1: Execute the Initial T-SQL Script

1. Open the T-SQL script **71 - Lab Exercise 3.sql**, and 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 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.



Note: What changes would you make to change the sort order to descending?

Results: After this exercise, you should have an understanding of how comments can be specified inside T-SQL scripts. You will also have an appreciation of how to order the results of a query.

Module Review and Takeaways

In this module, you have learned how 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.

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

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

You can use the SELECT statement to query tables and views. It is likely that you will use the SELECT statement more than any other single statement in T-SQL. You can manipulate the data with SELECT to customize how SQL Server returns the results. 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 table and column 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, in addition to enhancements that will add functionality and readability to your queries.

Lesson Objectives

At the end of this lesson, you will be able to:

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

Elements of the SELECT Statement

The SELECT and FROM clauses are the primary focus of this module. You will learn about the other clauses in later modules of this course. You have already learned the order of operations in logical query processing; this will help you to understand how to form your SELECT statements correctly.

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

For more information on the SELECT elements, see:

 **SELECT (Transact-SQL)**

<http://aka.ms/vvwmme>

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

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 can add others in the form of calculated expressions.

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 table or view name resolution, it is best to include the schema and object name, in the format SCHEMA.OBJECT—for example Sales.Customer.

If the table or view name contains irregular characters, such as spaces or other special characters, you 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 2016, semicolons are an optional terminator for most statements. However, future versions will require its use. For 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.

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

Keyword	Expression
SELECT	<select list>
FROM	<table or view>

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

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 you have defined them 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.

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 the asterisk is suitable for a quick test, avoid using it 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. Furthermore, returning data that is not needed can slow down your queries and cause performance issues if the source table contains a large number of rows.

By using an explicit column list in your SELECT clause, you will always achieve the desired results, providing 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.

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

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

- Displaying only specified columns

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

ACT 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 and manipulations can change the source column data, and use built-in T-SQL functions, which you will learn about later in this course.

As 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—they must return only a single value.

Calculated expressions can 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, pass the name of the column to the function as an input:

Create a Calculated 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 and other functions 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 just once at the time of query execution, and reuse the value for each row. This will be discussed later in the 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, (qty * unitprice)
FROM Sales.OrderDetails;
```

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **D:\Demofiles\Mod03\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. At the command prompt, type **y**, and press Enter. When the script has completed, press any key.
5. Start SQL Server Management Studio and connect to the **Azure SQL** database engine instance using SQL Server authentication.
6. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod03\Demo** folder.
7. In Solution Explorer, expand **Queries**, and open the **Demonstration A.sql** script file. You may need to enter your password to connect to the **Azure SQL** database engine.
8. In the **Available Databases** list, click **AdventureWorksLT**.
9. Select the code under the comment **Step 2**, and then click **Execute**.
10. Select the code under the comment **Step 3**, and then click **Execute**.
11. Select the code under the comment **Step 4**, and then click **Execute**.
12. Select the code under the comment **Step 5**, and then click **Execute**.
13. Select the code under the comment **Step 6**, and then click **Execute**.
14. Select the code under the comment **Step 7**, and then click **Execute**.
15. On the **File** menu, click **Close**.
16. 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 how 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. The rows are not 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).

- SQL query results are not truly relational:
 - Rows are not guaranteed to be unique
 - No guaranteed order
- Even unique rows in a source table can return duplicate values for some columns

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

country

Argentina
Argentina
Belgium
Austria
Austria

For example, consider a query that returns 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 easily 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
Austria
Brazil
Spain
France
Sweden
...
Germany
France
Finland
Poland
(91 rows(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;
```

Without further instruction, the query will return one result for each row in the Sales.Customers table; however, as only the country column is specified, you will see just this column 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.

- DISTINCT 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 before 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, you can replace the silent ALL default with DISTINCT, to eliminate the duplicate values:

DISTINCT Statement

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

This will return 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
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. After you have learned them, you could consider the relative performance costs of filtering with SELECT DISTINCT, compared to those other methods.

SELECT DISTINCT Syntax

Remember that DISTINCT looks at rows in the output set, created by the SELECT clause.

Therefore, only unique column values will be returned by a SELECT DISTINCT clause.

```
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

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 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 columns which, in this case, is the same seven employees.

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. In Solution Explorer, open the **Demonstration B.sql** script file. You may need to enter your password to connect to the **Azure SQL** database engine.
2. In the **Available Databases** list, click **AdventureWorksLT**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. On the **File** menu, click **Close**.
7. 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?

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. You can relabel columns by using aliases in the SELECT clause. However, columns created with expressions will not be named automatically. Column aliases can be used to provide custom column headers. At the table level, you can use aliases 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.

Use 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" can 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.

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 Equals Sign

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

• 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;
```

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
10248	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 list columns carefully, separating them with commas and adopting the AS style of aliases, to make it easier to spot mistakes.

Question: Which of the following statements use correct column aliases?

1. SELECT Name AS ProductName FROM Production.Product
2. SELECT Name = ProductName FROM Production.Product
3. SELECT ProductName == Name FROM Production.Product
4. SELECT ProductName = Name FROM Production.Product
5. SELECT Name AS Product Name FROM Production.Product

Use Aliases to Refer to Tables

Aliases can also be used in the FROM clause to refer to a table; this 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.

You can 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 can 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.

As 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 be useful.

- Create table aliases in the FROM clause
- Create table aliases with AS

```
SELECT custid, orderdate
FROM SalesOrders AS SO;
```

- Create table aliases without AS

```
SELECT custid, orderdate
FROM SalesOrders SO;
```

- Using table aliases in the SELECT clause

```
SELECT SO.custid, SO.orderdate
FROM SalesOrders AS SO
```

The Impact of Logical Processing Order on Aliases

When using column aliases, an issue can arise. 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 (HAVING and WHERE clauses will be covered in a separate module). An exception to this is the ORDER BY clause.

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;
```

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 is:

```
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 might 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 might 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'.
```

- 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

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. In Solution Explorer, open the **Demonstration C.sql** script file. You may need to enter your password to connect to the **Azure SQL** database engine.
2. In the **Available Databases** list, click **AdventureWorksLT**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. On the **File** menu, click **Close**.
8. 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 of a column 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 that is 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 (and other) clauses to return the result of an expression. The results appear as a calculated column and, for clarity, should be aliased.

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.

- T-SQL CASE expressions return a single (scalar) value
- CASE expressions may be used in:
 - SELECT column list
 - WHERE or HAVING clauses
 - ORDER BY clause
- CASE returns result of expression
 - Not a control-of-flow mechanism
- In SELECT clause, CASE behaves as calculated column requiring an alias

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:

- If a match is found, the first matching value is returned as the result of the CASE expression. Multiple matches are not permitted.

- 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

- If no match is found, a CASE expression returns the value found in an ELSE clause, if one exists.
- If no match is found and no ELSE clause is present, the CASE expression returns a NULL.

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; instead, it's a substitution using a single table:

CASE Expression

```
SELECT productid, productname, categoryid,
       CASE categoryid
         WHEN 1 THEN 'Beverages'
         WHEN 2 THEN 'Condiments'
         WHEN 3 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 the SQL Server 2016 Technical Documentation:

 **CASE (Transact-SQL)**

<http://aka.ms/ims4v6>

Demonstration: Simple CASE Expressions

In this demonstration, you will see how to:

- Use a simple CASE expression.

Demonstration Steps

Use a Simple CASE Expression

1. In Solution Explorer, open the **Demonstration D.sql** script file. You may need to enter your password to connect to the **Azure SQL** database engine.
2. In the **Available Databases** list, click **AdventureWorksLT**.
3. Select the code under the comment **Step 2**, and then click **Execute**.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Close SQL Server Management Studio, without saving changes.

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?

Lab: Writing Basic SELECT Statements

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server 2016. You can use your set of business requirements for data to 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: **20761B-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, so you can better understand different business requirements. After an initial overview, you will provide a report for the marketing department, whose staff want 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 ADVENTUREWORKS Database in Object Explorer
3. Write a Simple SELECT Statement That Returns All Rows and Columns from a Table
4. Write a SELECT Statement That Returns Specific Columns

► Task 1: Prepare the Lab Environment

1. Ensure that the 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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 ADVENTUREWORKS 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. Look at the names of the tables in the Sales schema.

► **Task 3: Write a Simple SELECT Statement That Returns All Rows and Columns from a Table**

1. Open the project file D:\Labfiles\Lab03\Starter\Project\Project.ssmssln and the T-SQL script 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 move 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. You can use drag-and-drop functionality to move items like table and column names from Object Explorer to the query window. Write the same SELECT statement using the drag-and-drop functionality.

► **Task 4: Write a SELECT Statement That Returns 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\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 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 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\Lab Exercise 2 - Task 1 Result.txt.

► **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\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 understand how to return only the different (distinct) rows in the result set of a query.

Exercise 3: Using Table and Column Aliases

Scenario

After receiving the initial list of customers, the marketing department would like to have column titles that are more readable 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 Table and Column Aliases
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 Lab Exercise 3.sql, and 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 are 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\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, 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\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 Table and Column Aliases**

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\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 to avoid such errors when using aliases for columns?

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

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 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\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\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\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

In this module, you have learned how to:

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



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?

Real-world Issues and Scenarios

You can create a column alias without using the AS keyword, something you are likely to see in code samples online, or written by developers you work with. While the T-SQL engine will parse this without issue, there is a problem when a comma is omitted between column names—the first column will take the name of the second column as its alias. Not only will the column have a misleading name, but you will also have one column too few in your result set. Always use the AS keyword to avoid this problem.

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. You have already 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 write queries containing joins, which allow you to retrieve data from two (or more) tables, based on data relationships between the tables.

In this module, you will learn how to write queries that combine data from multiple sources in Microsoft SQL Server 2016.

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 multitable queries.

The FROM Clause and Virtual Tables

You have already 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 holds true whether you are querying a single table or bringing together multiple tables as the source of your query. To learn about the additional capabilities of the FROM clause, it is 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.

- 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



Reader Aid: 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>;
```

You have learned 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 these aliases 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 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 multitable queries.

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

Join Terminology: Cartesian Product

When learning about writing multitable 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 6×2 . 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. With no 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.

- Characteristics of a Cartesian product
 - Output or intermediate result of FROM clause
 - Combine all possible combinations of two sets
 - In T-SQL queries, usually not desired
 - Special case: table of numbers

Name	Product	Name	Product
Devis	Alice Mutton	Davis	Alice Mutton
Funk	Crab Meat	Davis	Crab Meat
King	Ipoh Coffee	Funk	Ipoh Coffee

 **Reader Aid:** 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:

- 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.



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

- 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

T-SQL Syntax Choices

Throughout the history of SQL Server, the product has changed to keep pace with variations in the American National Standards Institute (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, in addition to any other filtering, it is not recommended here. Additionally, if a WHERE clause is accidentally omitted, ANSI SQL-89-style joins can easily become Cartesian products and cause performance problems.

- ANSI SQL-92

• Tables joined by JOIN operator in FROM Clause

```
SELECT ...
FROM Table1 JOIN Table2
ON <on_predicate>
```

- ANSI SQL-89

• Tables joined by commas in FROM Clause

• Not recommended: accidental Cartesian products!

```
SELECT ...
FROM Table1, Table2
WHERE <where_predicate>
```

The following queries illustrate this syntax and the 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;
```

 **Reader Aid:** 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.

Demonstration: Understanding Joins

In this demonstration, you will see how to:

- Use joins.

Demonstration Steps

Use Joins

1. Ensure that the 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **D:\Demofiles\Mod04\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. At the command prompt, type **y**, and then press Enter. When the script has completed, press any key.

5. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
6. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod04\Demo** folder.
7. In Solution Explorer, expand **Queries**, and then double-click the **11 - Demonstration A.sql** script file.
8. Select the code under the comment **Step 1**, and then click **Execute**.
9. Select the code under the comment **Step 2**, and then click **Execute**.
10. Select the code under the comment **Step 3**, and then click **Execute**.
11. Select the code under the comment **Step 4**, and then click **Execute**.
12. Select the code under the comment **Step 5**, and then click **Execute**.
13. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Question
<p>You have the following T-SQL query:</p> <pre>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;</pre>
<p>Which of the following types of join will the query perform?</p>
<p>Select the correct answer.</p>
<input type="checkbox"/> A cross join
<input type="checkbox"/> An inner join
<input type="checkbox"/> An outer left join
<input type="checkbox"/> An outer right join

Lesson 2

Querying with Inner Joins

In this lesson, you will learn how to write inner join queries, the most common type of multitable 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. As you have 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.

- 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

MCT USE ONLY. STUDENT USE PROHIBITED

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. 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 "o" 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.

- List tables in FROM Clause separated by JOIN operator
- Table aliases preferred
- Table order does not matter

```
FROM t1 JOIN t2
  ON t1.column = t2.column
```

```
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.
```

- 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 tables and the third table.

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;
```

- Join based on single matching attribute

```
SELECT ...
FROM Production.Categories AS C
JOIN Production.Products AS P
ON C.categoryid = P.categoryid;
```

- Join based on multiple matching attributes (composite join)

```
-- List cities and countries where both --
customers and employees live
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;
```

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;
```



Reader Aid: 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. The SQL Server query optimizer determines the order in which 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.

Demonstration Steps

Use Inner Joins

1. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. Select the code under the comment **Step 6**, and then click **Execute**.
8. 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?

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 multitable 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 also 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.
- Write queries using outer joins.
- Combine an OUTER JOIN operator in a FROM clause with a nullability test in a WHERE clause to reveal nonmatching 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.

First, 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 AdventureWorks2016 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.

- 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 nonmatched 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

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 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.
- Multitable 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.

- Return all rows from first table, only matches from second:
`FROM t1 LEFT OUTER JOIN t2 ON
t1.col = t2.col`
- Return all rows from second table, only matches from first:
`FROM t1 RIGHT OUTER JOIN t2 ON
t1.col = t2.col`
- Return only rows from first table, with no match in second:
`FROM t1 LEFT OUTER JOIN t2 ON
t1.col = t2.col
WHERE t2.col IS NULL`

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:

- 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 who 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;
```

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;
```

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. In Solution Explorer, open the **31 - Demonstration C.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. Select the code under the comment **Step 6**, and then click **Execute**.
8. Select the code under the comment **Step 7**, and then click **Execute**.
9. 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?	
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;

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.

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 from you.

- 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 nonmatching 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

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:

- 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;
```

Cross Join Example

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

Understanding Self Joins

So far, 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 in a table with one another. As you have seen, it's fairly easy to compare columns in 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.

To accomplish tasks like this, you should 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:

- 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;
```

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 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. In Solution Explorer, open the **41 - Demonstration D.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. 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?

Lab: Querying Multiple Tables

Scenario

You are an Adventure Works business analyst who will be writing reports using corporate databases stored in SQL Server 2016. You have been given 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: **20761B-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa\$\$w0rd**

Exercise 1: Writing Queries That Use Inner Joins

Scenario

You no longer need the supplied mapping information between categoryid and categoryname 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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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 who 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 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;
```

Execute the query exactly as written inside a query window and observe the result.

3. An error is shown. 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.
3. Change the prefix of the columns in the SELECT statement with full source table names and execute the statement.
4. There is 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 T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the TSQL database.
2. 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 the 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.

DO NOT USE PROHIBITED

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?

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

MCT USE ONLY. STUDENT USE PROHIBITED

► **Task 1: Execute the T-SQL Statement**

1. Open 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 give a more realistic example for a cross join in the next task.

► **Task 2: Write a SELECT Statement That Uses a Cross Join**

1. 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.
2. 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.
3. How many rows are 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**

- 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

In this module, you have learned how 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.



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?

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	5-20
Module Review and Takeaways	5-25

Module Overview

In this module, you will learn how to enhance a query to limit the number of rows that the query returns, 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. Therefore, if you require the output of a query to be displayed in a certain order, you should add an ORDER BY clause to your SELECT statement. In this module, you will learn how to write a query using ORDER BY to control the display order.

You have already 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, in addition to 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 the rows selected with a WHERE clause based on a predicate; you can also limit the number of rows with a TOP, or OFFSET and FETCH clause, based on the order of the rows selected.

When you work with real-world data in 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.

Objectives

Filter data with predicates in the WHERE clause:

- Sort data using ORDER BY.
- Filter data in the SELECT clause with TOP.
- Filter data with OFFSET and FETCH.

Lesson 1

Sorting Data

In this lesson, you will learn how to add an ORDER BY clause to a query to control the order of rows displayed in the output of the query.

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 be executed. ORDER BY enables you to control the sorting of rows as they are output from the query to the client application. Without an ORDER BY clause, SQL Server does not guarantee the order of rows—in keeping with relational theory.

To sort the output of your query, you will add an ORDER BY clause in this form:

- 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

ORDER BY Clause

```
SELECT      <select_list>
FROM        <table_source>
ORDER BY    <order_by_list> [ASC|DESC];
```

ORDER BY can take several types of element in its list:

Columns by name. Additional columns beyond the first specified in the list will be used as tiebreakers for nonunique 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, because of 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 included in the SELECT list.



Note: ORDER BY may also include a COLLATE clause, which provides a way to sort by a specific character collation, instead of the collation of the column in the table. Collations will be discussed further 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 by using ASC for ascending (A-Z, 0-9) or DESC for descending (Z-A, 9-0). 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 to least recent hire, 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;
```

For additional information on the ORDER BY clause, see the SQL Server 2016 Technical Documentation 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.

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;
```

• 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 <alias1>, <alias2>;
```

• Specifying sort order in the ORDER BY clause:

```
SELECT <column> AS <alias>
FROM <table source>
ORDER BY <column_name|alias> ASC|DESC;
```

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 Using Column 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.

A query against the Sales.Orders table, sorting the results by the orderdate column:

- 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;
```

ORDER BY Example 1

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate;
```

A query against the Sales.Orders table that sorts the output in descending order of orderdate (that is, most recent to oldest):

ORDER BY Example 2

```
SELECT orderid, custid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

A query against the HR.Employees table that sorts 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 3

```
SELECT hiredate, firstname, lastname
FROM HR.Employees
ORDER BY hiredate DESC, lastname ASC;
```

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

- Ensure that the MSL-TMG1, 20761B-MIA-DC, and 20761B-MIA-SQL virtual machines are running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
- Start SQL Server Management Studio and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication.
- Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod05\Demo** folder.
- In Solution Explorer, expand **Queries**, and then double-click **11 - Demonstration A.sql**.
- In the **Available Databases** list, click **ADVENTUREWORKSLT**.
- Select the code under the comment **Step 1**, and then click **Execute**.
- Select the code under the comment **Step 2**, and then click **Execute**.
- Select the code under the comment **Step 3**, and then click **Execute**.
- Select the code under the comment **Step 4**, and then click **Execute**.
- Select the code under the comment **Step 5**, and then click **Execute**.
- Select the code under the comment **Step 6**, and then click **Execute**.
- Keep SQL Server Management Studio open for the next demonstration.

Question: 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. Why is this?

NOTICE ONLY. STUDENT USE PROHIBITED

Lesson 2

Filtering Data with Predicates

When querying SQL Server, you will mostly want to retrieve only a subset of all rows stored in the table(s) 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

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.

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. (The evaluation may be NULL 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 SELECT clause of the query cannot be used in the WHERE clause predicate.
- Remember that, logically, the WHERE clause is the next phase in query execution after FROM, so the WHERE clause 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 the expressions for use in the WHERE clause.

- 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

For example, the following query, which uses a simple calculated expression in the SELECT list, will execute successfully:

Filtering Example

```
SELECT orderid, custid, YEAR(orderdate) AS ordyear
FROM Sales.Orders
WHERE YEAR(orderdate) = 2006;
```

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 positive 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, further improving performance.

WHERE Clause Syntax

In the SQL Server 2016 Technical Documentation, 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>
```

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.

- 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';
```

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.

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 logical operators to search within a range of dates:

Range Example

```
SELECT orderid, custid, orderdate
  FROM Sales.Orders
 WHERE orderdate >= '20070101' AND orderdate <= '20080630';
```

The following example accomplishes the same results using the BETWEEN operator:

BETWEEN Operator

```
SELECT orderid, custid, orderdate
  FROM Sales.Orders
 WHERE orderdate BETWEEN '20070101' AND '20080630';
```



Note: The use of comparison operators with date and time data types requires special consideration. For more information, see Module 6.

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. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. In the **Available Databases** list, click **ADVENTUREWORKSLT**.
3. Select the code under the comment **Step 1**, and then click **Execute**.
4. Select the code under the comment **Step 2**, and then click **Execute**.
5. Select the code under the comment **Step 3**, and then click **Execute**. Note the error message.
6. Select the code under the comment **Step 4**, and then click **Execute**.
7. Select the code under the comment **Step 5**, and then click **Execute**.
8. Select the code under the comment **Step 6**, and then click **Execute**.
9. Select the code under the comment **Step 7**, and then click **Execute**.
10. Select the code under the comment **Step 8**, and then click **Execute**.
11. Select the code under the comment **Step 9**, and then click **Execute**.
12. Select the code under the comment **Step 10**, and then click **Execute**.
13. Select the code under the comment **Step 11**, and then click **Execute**.
14. 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 use a TOP option to limit ranges of rows in the SELECT clause. 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, in addition to filtering 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 as a percentage of all candidate rows.

The simplified syntax of the TOP option is as follows:

- TOP allows you to limit the number or percentage of rows returned by a query
- Works with ORDER BY clause to limit rows by sort order:
 - If ORDER BY list is not unique, results are not deterministic (no single correct result set)
 - Modify ORDER BY list to ensure uniqueness, or use TOP WITH TIES
- Added to SELECT clause:
 - SELECT TOP (N) | TOP (N) Percent
 - With percent, number of rows rounded up (nondeterministic)
 - SELECT TOP (N) WITH TIES
 - Retrieve duplicates where applicable (deterministic)
- TOP is proprietary to Microsoft SQL Server

TOP Option

```
SELECT TOP (N) <column_list>
FROM <table_source>
WHERE <search_condition>
ORDER BY <order list>;
```

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: 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 for 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.

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, its potential for unique values, and the requirements of the query you are writing.

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: For the purposes of row count, TOP (N) PERCENT will round up to the nearest integer.

For additional information about the TOP clause, see the SQL Server 2016 Technical Documentation 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 disadvantages:

- TOP is proprietary to T-SQL and SQL Server.
- TOP does not support skipping a range of rows.
- Because 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 the OFFSET-FETCH extension to the ORDER BY clause.

Like TOP, OFFSET-FETCH enables 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.

OFFSET-FETCH is supported in SQL Server 2012, 2014, and 2016.

For more information about the OFFSET-FETCH clause, see *Using OFFSET and FETCH to limit the rows returned* in the SQL Server 2016 Technical Documentation at:



ORDER BY Clause (Transact-SQL)

<http://go.microsoft.com/fwlink/?LinkId=402718>

OFFSET-FETCH Syntax

The syntax for the OFFSET-FETCH clause is as follows:

OFFSET-FETCH Clause

```
OFFSET { integer_constant |
offset_row_count_expression } { ROW | ROWS
}
[FETCH { FIRST | NEXT }
{integer_constant |
fetch_row_count_expression } { ROW | ROWS
} ONLY]
```

- 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
 - ONLY optional—makes meaning clearer to human reader
- 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]
```

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 1

```
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 enables a more natural syntax.

To ensure the accuracy of the results, 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 might supply the OFFSET value, in addition to 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 them use the AdventureWorks 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 2

```
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 3

```
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. In Solution Explorer, open the **31 - Demonstration C.sql** script file.
2. In the **Available Databases** list, ensure **ADVENTUREWORKSLT** is selected.
3. Select the code under the comment **Step 1**, and then click **Execute**.
4. Select the code under the comment **Step 2**, and then click **Execute**.
5. Select the code under the comment **Step 3**, and then click **Execute**.
6. Select the code under the comment **Step 4**, and then click **Execute**.
7. Select the code under the comment **Step 5**, and then click **Execute**.
8. Select the code under the comment **Step 6**, and then click **Execute**.
9. Select the code under the comment **Step 7**, and then click **Execute**.
10. Select the code under the comment **Step 8**, and then click **Execute**.
11. Keep SQL Server Management Studio open for the next demonstration.

MCT USE ONLY. STUDENT USE PROHIBITED

Check Your Knowledge

Question	
<p>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?</p>	
<p>Select the correct answer.</p>	
	<pre>SELECT ProductID, ProductName, ProductNumber FROM Sales.Products ORDER BY ProductName ASC OFFSET 60 ROWS FETCH NEXT 20 ROWS ONLY</pre>
	<pre>SELECT ProductID, ProductName, ProductNumber FROM Sales.Products ORDER BY ProductName ASC OFFSET 40 ROWS FETCH NEXT 20 ROWS ONLY;</pre>
	<pre>SELECT TOP (20) ProductID, ProductName, ProductNumber FROM Sales.Products ORDER BY ProductName ASC</pre>
	<pre>SELECT TOP (20) WITH TIES ProductID, ProductName, ProductNumber FROM Sales.Products ORDER BY ProductName ASC</pre>

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.

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, and 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.

- 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

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:

- 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.

- 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 <>

```
SELECT custid, city, region, country
FROM Sales.Customers
WHERE region IS NOT 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.

MCT USE ONLY. STUDENT USE PROHIBITED

Note the following example:

Incorrectly Testing for NULL

```
SELECT empid, lastname, region
FROM HR.Employees
WHERE region = NULL;
```

This returns unexpected results:

empid	lastname	region
(0 row(s) affected)		

- Use the IS NULL (or IS NOT NULL) operator rather than equal (or not equal).

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. In Solution Explorer, open the **41 - Demonstration D.sql** script file.
2. In the **Available Databases** list, ensure **ADVENTUREWORKSLT** is selected.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. Select the code under the comment **Step 6**, and then click **Execute**.
8. Select the code under the comment **Step 7**, and then click **Execute**.
9. Close SQL Server Management Studio.

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, and this requested action has been applied to the database.

Will the above query return these employees?

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: **20761B-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa\$\$w0rd**

Exercise 1: Write 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 Using a WHERE Clause
3. Write a SELECT Statement Using an IN Predicate in the WHERE Clause
4. Write a SELECT Statement Using 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 Customers Without Orders

► Task 1: Prepare the Lab Environment

1. Ensure that the 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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 Using 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 Using 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 Using 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 = N'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 files D:\Labfiles\Lab05\Solution\55 - Lab Exercise 1 - Task 4a Result.txt and D:\Labfiles\Lab05\Solution\56 - Lab Exercise 1 - Task 4b 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 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\57 - 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.

Exercise 2: Write 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. From previous modules, remember 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 Using 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 Using an ORDER BY Clause

1. Open the T-SQL script **61 - Lab Exercise 2.sql**, and 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. There is 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 3a and 3b 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: Write Queries that Filter Data Using the TOP Option

Scenario

The sales department wants 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. 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.Orders** table, and retrieve the **orderid** and **orderdate** columns. Retrieve the 20 most recent orders, ordered by **orderdate**.
3. Execute the written statement and compare the results that you achieved with the recommended result shown in the file **72 - Lab Exercise 3 - Task 1 Result.txt**.

► Task 2: Use the OFFSET-FETCH Clause to Implement the Same Task

1. Write a SELECT statement to retrieve the same result as in task 1, but use the OFFSET-FETCH clause.
2. Execute the written statement and compare the results that you achieved with the results from task 1.
3. Compare the results that you achieved with the recommended result shown in the file **73 - Lab Exercise 3 - Task 2 Result.txt**.

► Task 3: Write a SELECT Statement to Retrieve the Most Expensive Products

1. Write a SELECT statement to retrieve the **productname** and **unitprice** columns from the **Production.Products** table.
2. Execute the T-SQL statement and notice the number of the rows returned.
3. Modify the SELECT statement to include only the top 10 percent of products based on **unitprice** ordering.
4. Execute the written statement and compare the results that you achieved with the recommended result shown in the file **74 - Lab Exercise 3 - Task 3 Result.txt**. Notice the number of rows returned.
5. Is it possible to implement this task with the OFFSET-FETCH clause?

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.

Exercise 4: Write 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 T-SQL script **81 - Lab Exercise 4.sql**, and 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

- You are given the parameters **@pagenum** for the requested page number and **@pagesize** for the requested page size. Can you work out how to write a generic form of the OFFSET-FETCH clause using those parameters? (Don't worry about not being familiar with those parameters yet.)

Results: After this exercise, you will be able to use OFFSET-FETCH to work page-by-page through a result set returned by a SELECT statement.

Question: What is the difference between filtering using the **TOP** option, and filtering using the **WHERE** clause?

Module Review and Takeaways

In this module, you have learned how to enhance a query to limit the number of rows that the query returns, and control the order in which the rows are displayed.

Review Question(s)

Question: Does the physical order of rows in a 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

Module 6

Working with SQL Server 2016 Data Types

Contents:

Module Overview	6-1
Lesson 1: Introducing SQL Server 2016 Data Types	6-2
Lesson 2: Working with Character Data	6-11
Lesson 3: Working with Date and Time Data	6-20
Lab: Working with SQL Server 2016 Data Types	6-26
Module Review and Takeaways	6-32

Module Overview

To write effective queries in T-SQL, you should understand how SQL Server® 2016 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 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 data 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 SQL Server data types, type precedence, and type conversions.
- Write queries using numeric data types.
- Write queries using character data types.
- Write queries using date and time data types.

Lesson 1

Introducing SQL Server 2016 Data Types

In this lesson, you will explore many of the data types SQL Server uses to store data, and learn how data is converted between data 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 might wish to note which data types are used in your environment. You can then plan your reports and client applications with sufficient capacity to display the range of values held by the SQL Server data types. You may also need to plan for data type conversions in your queries to display SQL Server data in other environments.

If your focus is to continue into database development and administration, you might wish to note 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 2016 defines a set of system data types for storing data in columns, holding values temporarily in variables, operating on data in expressions, and passing parameters to 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, along with designing tables and creating other objects.

Developers might 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; however, this lesson will focus on the built-in system data types.

- SQL Server associates columns, expressions, variables and parameters with data types
- Data types determine the kind of data that can be held in a column or variable
 - Integers, characters, dates, decimals, binary strings, and so on
- SQL Server supplies built-in data types
- Developers can also define custom data types

SQL Server Data Type Categories	
Exact numeric	Unicode character strings
Approximate numeric	Binary strings
Date and time	Other
Character strings	

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—whole numbers with varying degrees of capacity.
 - Decimals—decimal numbers with control over both the total number of digits stored and the number of digits to the right of the decimal place.
- **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 byte streams or hashes, to support custom applications.
- **Other data types.** This catch-all category includes several special types that fall outside the other categories. Some of these data types can be used as column data types (and are therefore accessible to queries). This category also includes data types not used for storage, but rather for special operations, such as cursor manipulation or creating table variables for further processing. If you are a report writer, you may only encounter the data types used for columns, such as the **uniqueidentifier** and **xml** data types.

As you learn about these types, take note of the relationship between capacity and storage requirements.

Numeric Data Types

- Numeric data types fall into one of two subcategories—exact numeric and approximate numeric.
- Exact numeric data types:
 - Integer data types. The distinction between the integer data types (**tinyint**, **smallint**, **int**, **bigint**) relates to their capacity and storage requirements. The **tinyint** data type, for example, holds values from 0 to 255 with a storage cost of 1 byte. By contrast, the **bigint** data type holds values from -2^{63} (-9,223,372,036,854,775,808) to $2^{63}-1$ (9,223,372,036,854,775,807) with a storage cost of 8 bytes.
 - Decimal data types. These data types are specified with the total number of digits to be stored (precision) and the number of digits to the right of the decimal place (scale). The larger the precision, the greater the storage cost. Note that there is no functional difference between the **decimal** data type and **numeric** data type—**decimal** is the ISO standards-compliant name for the data type; **numeric** is used for backward compatibility with earlier versions of SQL Server.
 - Money data types, for storing monetary or currency values with a scale of up to four decimal places. As with the integer types, the distinction between the money data types **money** and **smallmoney** relates to their capacity and storage requirements. The **smallmoney** data type holds values from -214,748.3648 to 214,748.3647 with a storage cost of 4 bytes. The **money** data type holds values from -922,337,203,685,477.5808 to 922,337,203,685,477.5807 with a storage cost of 8 bytes.

Exact Numeric Data Types		
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} to $2^{63}-1$ (+/- 9 quintillion)	8
bit	0, 1, 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

- Boolean data type. The **bit** data type is used to store Boolean values (true/false) which are treated by SQL Server as numeric values—1 for true and 0 for false.

For more information, see the following topics in the SQL Server 2016 Technical Documentation:

Data Types (Transact-SQL) at:

 **Data Types (Transact-SQL)**

<http://aka.ms/we8bzv>

Precision, Scale, and Length (Transact-SQL)

 **Precision, Scale, and Length (Transact-SQL)**

<http://aka.ms/t0hwx5>

decimal and numeric (Transact-SQL)

 **decimal and numeric (Transact-SQL)**

<http://aka.ms/sqkh78>

- Approximate numeric data types. 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, because of a lack of precision, loses accuracy.
 - The **float** data type takes an optional parameter of the number of bits used to store the mantissa of the **float** number in scientific notation. The size of the mantissa value 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.
 - The **real** data type is a synonym for a **float** data type with a mantissa value of 24 (that is, **float(24)**)



Note: Note that, in this context, the term *mantissa* is used to mean the significant digits of the floating point number. In mathematics, this portion of the number is more commonly referred to as the *significand*; however, in computer science, it is commonly referred to as the *mantissa*.

See the topic *float and real (Transact-SQL)* in the SQL Server 2016 Technical Documentation:

 **float and real (Transact-SQL)**

<http://aka.ms/noqiea>

Binary String Data Types

Binary string data types allow a developer to store binary information, such as serialized files, images, byte streams, and other specialized data. If you are considering using the binary data type, note the differences in range and storage requirements, compared with numeric and character string data. You can choose between fixed-width and variable-width binary strings; the differences between these will be explained in the character data type lesson later in the module.

The following example shows an integer value being converted to a binary data type:

Converting to Binary Data Type

```
SELECT CAST(12345 AS binary(4)) AS Result;
```

Returns the following:

```
Result
-----
0x00003039
```

The two leading characters in the output (0x) indicate that the output is a binary string.

For more information, see the *binary and varbinary (Transact-SQL)* topic in the SQL Server Technical Documentation:

binary and varbinary (Transact-SQL)

<http://aka.ms/o0ap4l>

 **Note:** The **image** data type is also a binary string type but is marked for removal in a future version of SQL Server. **varbinary(max)** should be used instead.

Other Data Types

In addition to numeric and binary types, SQL Server also supplies some other data types for specialized use cases, such as storage and processing of XML, generation and storage of globally unique identifiers (GUIDs), the representation of hierarchies, and more:

- The **xml** data type allows the storage and manipulation of Extensible Markup Language data (XML). The advantage of the **xml** data type over storing XML in a character data type is that the **xml** data type allows XML nodes and attributes to be queried within a T-SQL query using XQuery expressions. The **xml** data type also optionally allows an XML schema to be enforced. Each instance of an **xml** data type can store up to 2 GB of data.

• Binary string data types

Data Type	Range	Storage (bytes)
binary(n)	1 to 8000 bytes	n bytes
varbinary(n)	1 to 8000 bytes	n bytes + 2
varbinary(max)	1 to 2.1 billion (approx.) bytes	n bytes + 2

The **image** data type is also a binary string type but is marked for removal in a future version of SQL Server; **varbinary(max)** should be used instead

Data type	Range	Storage (bytes)	Remarks
xml	0-2 GB	0-2 GB	Stores XML in native Hierarchical structure
uniqueidentifier	Auto-generated	16	Globally unique identifier (GUID)
hierarchyid	n/a	Depends on content	Represents position in a hierarchy
rowversion	Auto-generated	8	Previously called timestamp
geometry	0-2 GB	0-2 GB	Shape definitions in Euclidian geometry
geography	0-2 GB	0-2 GB	Shape definitions in round-earth geometry
sql_variant	0-8000 bytes	Depends on content	Can store data of various other data types in the same column
cursor	n/a	n/a	Not a storage datatype—used for cursor operations
table	n/a	n/a	Not a storage data type—used for query operations



Additional Reading: See course 20472-2: *Developing Microsoft SQL Server Databases* for additional information on the XML data type.

- The **uniqueidentifier** data type allows the generation and storage of globally unique identifiers (GUIDs), stored as a 16-byte value. Values for the **uniqueidentifier** data type can be generated within SQL Server by using the NEWID() system function; they can also be generated by external applications or converted from string values.

The following example demonstrates the various methods that can be used to generate a GUID:

Creating GUIDs for the uniqueidentifier Data Type

```
SELECT NEWID() AS GUID_from_NEWID, CAST('1C0E3B5C-EA7A-41DC-8E1C-D0A302B5E58B' AS
uniqueidentifier) AS GUID_cast_from_string;
```

Returns:

GUID_from_NEWID	GUID_cast_from_string
DB71DBAE-460B-41DD-8CF1-FBEE3000BE0D	1C0E3B5C-EA7A-41DC-8E1C-D0A302B5E58B

- The **hierarchyid** data type is used to simplify the recording and querying of hierarchical relationships between rows in the same table—for example, the levels in an organizational chart or a bill of materials. SQL Server stores **hierarchyid** as a variable-length binary data type; the hierarchy is exposed through built-in functions.



Additional Reading: See course 20472-2: *Developing Microsoft SQL Server Databases* for additional information on the **hierarchyid** data type.

- The **rowversion** data type stores an automatically generated 8-byte binary value in a table that increments each time a row is inserted or updated. **Rowversion** values do not store date or time information, but can be used to detect whether a row has been changed since it was last read by a client application (for instance, when implementing optimistic locking).
- The spatial data types are special complex data types for dealing with geometric and geographic data. A detailed discussion of these types is beyond the scope of this course:
 - The **geometry** data type is used to store data in a Euclidean (flat) coordinate system. Arrays of coordinates defining lines, polygons and other simple geometric shapes can be stored in the **geometry** data type. Special built-in methods are available for carrying out operations on **geometry** data.
 - The **geography** data type is used to store data in a round-earth coordinate system, such as GPS latitude and longitude coordinates. As with the **geography** data type, shape definitions can be stored in the **geography** type, then built-in methods used to operate on **geography** data.
- The **sql_variant** type is a special type that may be used to store data of any other built-in data type—for instance, enabling integer, decimal and character data to be stored in the same column. Use of the **sql_variant** data type is not a best practice for typical database designs, and its use may indicate design problems. The **sql_variant** data type is listed here for completeness.

The following data types may not be used as data types for columns in tables or views; they are used as variables or parameters for stored procedures:

- The **cursor** data type is used to reference a cursor object, which allows row-by-row processing of a data set. A discussion of cursors is beyond the scope of this module.
- The **table** data type is used to define a table variable or stored procedure parameter, which has many of the properties of a standard database table but exists only in the context of the session for which it was created. Table data types are typically used to temporarily store the results of T-SQL statements for further processing later. You will learn about uses for the **table** data type later in this course.

For information on all of SQL Server's data types, see the SQL Server 2016 Technical Documentation, starting from:

Data Types (Transact-SQL)

<http://aka.ms/we8bzy>

Data Type Precedence

When combining or comparing different data types in your queries, such as in a WHERE or JOIN 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 attempt to 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
6. int
7. tinyint
8. nvarchar
9. char

- Data type precedence determines which data type will be chosen when expressions of different types are combined
- By default, the data type with the lower precedence is converted to the data type with the higher precedence
- It is important to understand implicit conversions
 - Conversion to a data type of lower precedence must be made explicitly (using CAST or CONVERT functions)
- Example precedence (low to high)
 - CHAR -> VARCHAR -> NVARCHAR -> TINYINT -> INT -> DECIMAL -> TIME -> DATE -> DATETIME2 -> XML
- Not all combinations of data type have a conversion (implicit or explicit)

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 data 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; therefore, if it fails (such as when your operation requires converting between data types for which no implicit conversion exists), you will need to explicitly convert the data type.

You will learn how to use the **CAST** and **CONVERT** functions for this purpose in the next module. There are some combinations of data types for which no conversion, explicit or implicit, is possible.

For more information and a complete list of data types and a list of precedence, see the SQL Server 2016 Technical Documentation:

 **Data Type Precedence (Transact-SQL)**

<http://aka.ms/a8ihqi>

For complete information on pairs of data types requiring implicit or explicit conversion, or for which no conversion is available, see the chart in the *Implicit Conversions* section of *CAST and CONVERT (Transact-SQL)*:

 **CAST and CONVERT (Transact-SQL) - Implicit Conversions**

<http://aka.ms/asaqq3>

When are Data Types Converted?

When querying SQL Server, there are a number of scenarios in which data might be converted between data types:

- When data is moved, compared to, or combined with other data.
- During variable assignment.
- When using any operator that involves operands of different types.
- When T-SQL code explicitly converts one data type to another, using the **CAST** or **CONVERT** function.

- Data type conversion scenarios
 - When data is moved, compared to or combined with other data
 - During variable assignment
- Implicit conversion
 - When comparing data of one data type to another
 - Transparent to the user
- Explicit conversion
 - Uses **CAST** or **CONVERT** functions

In the example in the previous topic, a variable of the **tinyint** data type was implicitly converted to an **int** data type when **tinyint** and **int** data types were added together in a query:

Implicit Conversion Example - Integer Data Types

```
DECLARE @myTinyInt AS tinyint = 25;
DECLARE @myInt as int = 9999;
SELECT @myTinyInt + @myInt;
```

You might also anticipate that an implicit conversion will take place in the following example:

Implicit Conversion Example - Integer and Character Data Types

```
DECLARE @myChar AS char(5) = '6';
DECLARE @myInt AS int = 1;
SELECT @myChar + @myInt;
```

Question: In the example, which data type will be converted? To which data type will it be converted?

As you have learned, SQL Server will automatically attempt to perform an implicit conversion from a lower-precedence data type to a higher-precedence data type.

Implicit data type conversion is transparent to the user, unless the conversion fails. See the following example:

Failing Implicit Conversion Example

```
DECLARE @myChar AS char(5) = 'six';
DECLARE @myInt AS int = 1;
SELECT @myChar + @myInt;
```

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 data type for the purposes of this query, you need to explicitly convert it. You will learn how to do this in the next module.

To learn more about data type conversions, see the SQL Server 2016 Technical Documentation at:



Data Type Conversion (Database Engine)

<http://aka.ms/t5db1j>

Demonstration: SQL Server Data Types

In this demonstration, you will see how to:

- Convert data types.

Demonstration Steps

Convert Data Types

- Ensure that the 20761B-MIA-DC, and 20761B-MIA-SQL virtual machines are running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
- Start SQL Server Management Studio and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication.
- Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod06\Demo** folder.

4. In Solution Explorer, expand **Queries**, and then double-click **11 - Demonstration A.sql**.
5. In the **Available Databases** list, click **AdventureWorksLT**.
6. Select the code under the comment **Step 2**, and then click **Execute**.
7. Select the code under the comment **Step 3**, and then click **Execute**. Note the error message.
8. Select the code under the comment **Step 4**, and then click **Execute**.
9. Keep SQL Server Management Studio open for the next demonstration.

Categorize Activity

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

Items	
1	tinyint
2	float
3	binary
4	int
5	real
6	varbinary
7	bigint
8	decimal
9	money
10	bit

Category 1	Category 2	Category 3
Exact Numeric Data Types	Approximate Numeric Data Types	Binary Data Types

Lesson 2

Working with Character Data

It is likely that the data you will work with in your T-SQL queries will include 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 might find useful in your queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the character data types supplied by SQL Server.
- Describe the impact of collation on character data.
- Concatenate strings.
- Extract and manipulate character data using built-in functions.
- Write queries using the LIKE predicate for matching patterns in character data.

Character Data Types

Even though SQL Server has many numeric data types, working with numeric data is relatively straightforward because numeric data follows a clearly defined set of mathematical rules.

By comparison, although there are fewer character data types available, working with character data in SQL Server can be more complicated. This is because you need to consider multiple languages, character sets, accented characters, sort rules and case sensitivity, and capacity and storage. Each of these factors might have an impact on which character data types you encounter when writing queries.

Character data types in SQL Server are categorized by two characteristics:

- Support for either fixed-width or variable-width data:
 - Fixed-width data is always stored at a consistent size, regardless of the number of characters in the character data. Any unused space is filled with padding.
 - Variable-width data is stored at the size of the character data, plus a small overhead.
- Support for either a single-byte character set or a multi-byte character set:
 - A single-byte character set supports up to 256 different characters, stored as one byte per character. By default, SQL Server uses the ASCII character set to interpret this data.
 - A multi-byte character set supports more than 65,000 different characters by storing each character as multiple bytes—typically two bytes per character, but sometimes more. SQL Server uses the UNICODE UCS-2 character set to interpret this data.

- SQL Server supports two kinds of character data as fixed-width or variable-width data:
 - Single-byte: **char** and **varchar**
 - One byte stored per character
 - Only 256 possible characters—limits language support
 - Multibyte: **nchar** and **nvarchar**
 - Multiple bytes stored per character (usually two bytes, but sometimes up to four)
 - More than 65,000 characters represented—multiple language support
 - Precede character string literals with N (National)
- **text** and **ntext** data types are deprecated, but may still be used in older systems
 - In new development, use **varchar(max)** and **nvarchar(max)** instead

The four available character data types support all possible combinations of these characteristics:

Data Type	Fixed Width?	Variable Width?	Single-Byte Characters?	Multi-Byte Characters?
char	Yes		Yes	
nchar	Yes			Yes
varchar		Yes	Yes	
nvarchar		Yes		Yes

Definitions for columns or variables take an optional value that defines the maximum length of the character data to be stored. You will almost always need to specify a value for the string length; if the maximum length value is not supplied, the default value is one character.

The **varchar** and **nvarchar** data types support the storage of very long strings of character data by using **max** for this value. Use of **varchar(max)** and **nvarchar(max)** replaces the use of the deprecated **text** and **ntext** types.

Data Type	Range	Storage
char(n)	1-8000 characters	n bytes, padded
nchar(n)	1-4000 characters	2*n bytes, padded
varchar(n)	1-8000 characters	Actual length + 2 bytes
nvarchar(n)	1-4000 characters	
varchar(max)	Up to 2 GB	Actual length + 2 bytes
nvarchar(max)		



Note: All character data is delimited with single quotation marks.

- Single-byte character data is indicated with single quotation marks alone—for example 'SQL Server'.
- Multi-byte character data is indicated by single quotation marks with the prefix N (for National)—for example N'SQL Server'. The N prefix is always required, even when inserting the data into a column or variable with a multi-byte type.

Collation

In addition to character byte count and length, SQL Server character data types are assigned a collation.

A collation is a collection of properties that determine several aspects of character data, including:

- Language or locale, from which is derived:
 - Character set
 - Sort order
- Case sensitivity
- Accent sensitivity

 **Note:** A default collation is configured during the installation of SQL Server, but can be overridden on a per-database or per-column basis. As you will see, you might 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, whether it is case-sensitive.

The following query will return different results, depending on whether the column being tested in the WHERE clause is case-sensitive or not:

If the column is case-sensitive, this query will return results. Note that the case of the search term matches the case of the data as stored in the database:

Case-Sensitivity Example (1)

```
SELECT empid, lastname
FROM HR.employees
WHERE lastname = N'Funk';
```

Amending the search term, so that the case no longer matches the data as stored in the database, would result in no rows being returned:

Case-Sensitivity Example (2)

```
SELECT empid, lastname
FROM HR.employees
WHERE lastname = N'funk';
```

The COLLATE clause can be used to override the collation of a column and force a different collation to be applied when the query is run.

This example forces a case-sensitive and accent-sensitive comparison using the Latin1_General sort rules and character table by adding a COLLATE clause to the WHERE clause:

Using COLLATE in the WHERE Clause

```
SELECT empid, lastname
FROM HR.employees
WHERE lastname COLLATE Latin1_General_CS_AS = N'Funk';
```

- Collation is a collection of properties for character data
 - Character set
 - Sort order
 - Case sensitivity
 - Accent sensitivity
- When querying, collation awareness is important for comparison
 - Is the database case-sensitive? If so:
 - 'Funk' does not equal 'funk'
 - SELECT * FROM HR.Employee does not equal SELECT * FROM HR.employees
- Add COLLATE clause to control collation comparison


```
SELECT empid, lastname
FROM HR.employees
WHERE lastname COLLATE Latin1_General_CS_AS = N'Funk';
```

NO TUTORIALS ONLY. STUDENTS USE PROHIBITED



Note: Note that database-level collation settings apply to database object names (such as tables and views) as well as to character data.

For example, in a database with a case-sensitive default collation, the table names

“HR.Employees” and “HR.employees” would refer to two different objects. In a database with a case-insensitive collation, the table names “HR.Employees” and “HR.employees” would refer to the same object.

For more information on this topic, see the SQL Server 2016 Technical Documentation:



COLLATE (Transact-SQL)

<http://aka.ms/ty97q8>



Collation and Unicode Support

<http://aka.ms/pm56d9>

String Concatenation

There are multiple ways to concatenate, or join together, multiple character data, or string, values in SQL Server.

The CONCAT function takes at least two (or more) data values as arguments and returns a string value with the input values concatenated together.

If any of the input data values is not of a character data type, it will be implicitly converted to a character data type.

Any NULL values will be converted to an empty string.

Syntax for the CONCAT function:

CONCAT Function Syntax

```
CONCAT ( string_value1, string_value2 [, string_valueN ] )
```

An example of the use of the CONCAT function:

Concatenating Strings Using CONCAT

```
SELECT custid, city, region, country,
CONCAT(city, ', ' + region, ', ' + country) AS location
FROM Sales.Customers;
```

Part of the result returned by this query is shown below:

custid	city	region	country	location
1	Berlin	NULL	Germany	Berlin, Germany
2	México D.F.	NULL	Mexico	México D.F., Mexico
3	México D.F.	NULL	Mexico	México D.F., Mexico
4	London	NULL	UK	London, UK
5	Luleå	NULL	Sweden	Luleå, Sweden

See the topic **CONCAT (Transact-SQL)** in the SQL Server 2016 Technical Documentation:

 **CONCAT (Transact-SQL)**

<http://aka.ms/b9c34t>

The **CONCAT** function was introduced in SQL Server 2012.

In earlier versions of SQL Server than 2012, the **CONCAT** function is not available; string concatenation is carried out using the **+** (plus) operator.

If any of the string values concatenated with the **+** operator is **NULL**, the output string will be **NULL**.

No conversion of data types is carried out (see note below).

The following example shows the use of the **+** operator to concatenate a given name, space, and family name into a single string:

Concatenating Strings Using **+**

```
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 of a numeric data type when concatenating. Character data types have a lower precedence than numeric data type, and SQL Server will attempt to convert and add mixed data types rather than concatenating them.

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:

- **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 shows the use of the **FORMAT** function to format a **money** value as currency in various locales:

FORMAT Function

```
DECLARE @m money = 120.595

SELECT @m AS unformatted_value,
FORMAT(@m, 'C', 'zh-cn') AS zh_cn_currency,
FORMAT(@m, 'C', 'en-us') AS en_us_currency,
FORMAT(@m, 'C', 'de-de') AS de_de_currency;
```

• Common functions that modify character strings

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 up to integer_value.
LEN, DATALENGTH	LEN (string, expression) DATALENGTH (expression)	LEN returns the number of characters in string_expression, excluding trailing spaces. DATALENGTH returns the number of bytes used.
CHARINDEX	CHARINDEX (expressionToFind, expressionToSearch)	Searches expressionToSearch for expressionToFind and returns its start position if found.
REPLACE	REPLACE (string, expression, string_pattern, string_replacement)	Replaces all occurrences of string_pattern in string_expression with string_replacement.
UPPER, LOWER	UPPER (character_expression) LOWER (character_expression)	UPPER converts all characters in a string to uppercase. LOWER converts all characters in a string to lowercase.

Returns:

```
unformatted_value zh_cn_currency en_us_currency de_de_currency
----- ----- -----
120.595     ¥120.60      $120.60      120,60 €
```

- **SUBSTRING**—allows you to return part of a character string given a starting point and a number of characters to return.

This example shows the use of SUBSTRING to return a portion of a string:

SUBSTRING Example

```
SELECT SUBSTRING('Microsoft SQL Server ',11,3) AS Result;
```

Returns:

```
Result
-----
SQL
```

- **LEFT** and **RIGHT**—allows you to return a number of characters from the left or right of a string.

This example shows the use of LEFT and RIGHT to select portions of a string:

LEFT and RIGHT Example

```
SELECT LEFT('Microsoft SQL Server',9) AS left_example,
RIGHT('Microsoft SQL Server',6) AS right_example;
```

Returns:

```
left_example right_example
----- -----
Microsoft      Server
```

- **LEN** and **DATALENGTH**—allows you to query metadata about the number of characters or the number of bytes stored in a string.

This example shows the result returned from LEN and DATALENGTH for the same padded string:

LEN and DATALENGTH Example

```
SELECT LEN('Microsoft SQL Server      ')      AS [LEN];
SELECT DATALENGTH('Microsoft SQL Server      ') AS [DATALENGTH];
```

Returns:

```
LEN
-----
20
DATALEN
-----
25
```

- **CHARINDEX**—allows you to query the start position of a string within another string. If the target string is not found, CHARINDEX returns 0.

This example shows output of CHARINDEX when the searched-for string is found:

CHARINDEX Example

```
SELECT CHARINDEX('SQL','Microsoft SQL Server') AS Result;
```

Returns:

```
Result
-----
11
```

- REPLACE—allows you to substitute one string for another within a target string.

This example shows the output of REPLACE when the searched-for string is found:

REPLACE Example

```
SELECT REPLACE('Learning about T-SQL string functions','T-SQL','Transact-SQL') AS Result;
```

Returns:

```
Result
-----
Learning about Transact-SQL string functions
```

- UPPER and LOWER—for performing character case conversions.

This example shows the use of UPPER and LOWER to manipulate the case of strings:

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, see the SQL Server 2016 Technical Documentation:

String Functions (Transact-SQL)

<http://aka.ms/l6hg9>

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 a 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 any other character than an 'A'.
- **ESCAPE** is used to set an escape character, meaning you can search for a character that is a wildcard character but to treat it as a literal, rather than a wildcard. Each instance of the special character to be treated as a literal must be preceded by the specified escape character. For example, LIKE N'10% off%' ESCAPE '!' will match any string that starts with '10% off', but would not match the string '100 special offers' (which would be matched if the ESCAPE character was not used).

- The LIKE predicate can be used to check a character string for a match with a pattern
- Patterns are expressed with symbols
 - % (Percent) represents a string of any length
 - _ (Underscore) represents a single character
 - [<List of characters>] represents a single character within the supplied list
 - [<Character> - <character>] represents a single character within the specified range
 - [^<Character list or range>] represents a single character not in the specified list or range
 - ESCAPE Character allows you to search for characters that would otherwise be treated as part of a pattern - %, _, [, and]

```
SELECT categoryid, categoryname, description
FROM Production.Categories
WHERE description LIKE 'Sweet%';
```

For further information on LIKE, see the SQL Server 2016 Technical Documentation:



LIKE (Transact-SQL)

<http://aka.ms/rm8ihw>

Demonstration: Working with Character Data

In this demonstration, you will see how to:

- Manipulate character data.

Demonstration Steps

Manipulate Character Data

1. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. In the **Available Databases** list, click **AdventureWorksLT**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3a**, and then click **Execute**.
5. Select the code under the comment **Step 3b**, and then click **Execute**.

6. Select the code under the comment **Step 4**, and then click **Execute**.
7. Select the code under the comment **Step 5**, and then click **Execute**.
8. Select all the code under the comment **Step 6**, and then click **Execute**.
9. Select the code under the comment **Step 7**, and then click **Execute**.
10. 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'?

Lesson 3

Working with Date and Time Data

Date and time data is very common when you are working with SQL Server data types. In this lesson, you will learn which data types are used to store date and time 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. As you may need to work with data created for older versions of SQL Server, even though you're writing queries for SQL Server 2016, it will be useful to review past support for date and time data:

- Before SQL Server 2008, there were only two data types for date and time data: **datetime** and **smalldatetime**. Each of these stored both the date and the time in a single value. For example, a **datetime** could store '20140212 08:30:00' to represent February 12, 2014 at 08:30.
- 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. For new work, Microsoft recommends these types over the older **datetime** and **smalldatetime**.
- 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).

For more information on all the date and time data types, see the SQL Server 2016 Technical Documentation:



Date and Time Types

<http://aka.ms/aeagy8>

• Older versions of SQL Server support only datetime and smalldatetime data types				
• SQL Server 2008 introduced date , time , datetime2 and datetimeoffset data types				
• SQL Server 2012 added further functionality for working with date and time data types				
Datatype	Storage (bytes)	Date Range (Gregorian Calendar)	Accuracy	Recommended Format
datetime	8	January 1, 1753 to December 31, 9999	Rounded to increments of .000, .003, or .007 seconds	YYYYMMDD hhmmss[.mmm]
smalldatetime	4	January 1, 1900 to June 6, 2079	1 minute	YYYYMMDD hhmmss[.mmm]
datetime2	6 to 8	January 1, 0001 to December 31, 9999	100 nanoseconds	YYYYMMDD hhmmss[.nnnnnn]
date	3	January 1, 0001 to December 31, 9999	1 day	YYYY-MM-DD
time	3 to 5	n/a – time only	100 nanoseconds	hhmmss[.nnnnnn]
datetimeoffset	8 to 10	January 1, 0001 to December 31, 9999	100 nanoseconds	YYYY-MM-DDThhmmss[.nnnnnn][(-+)]hhmm

Entering Date and Time Data Types Using Strings

To use date and time data in your queries, you will need to be able to represent date and time data in T-SQL. SQL Server doesn't offer the means to enter dates and times as literal values, so you will use character strings (often referred to as string literals) which are delimited, like all other strings in SQL Server, with single quotes. SQL Server will implicitly convert the string literals to date and time values. (You might also explicitly convert string literals with the T-SQL CAST and CONVERT functions, which you will learn about in the next module.)

- SQL Server doesn't offer a means to enter a date or time value as a literal value
 - Dates and times are entered as character literals and converted explicitly or implicitly
 - For example, `char` converted to `datetime` due to precedence
 - Formats are language-dependent, and can cause confusion
- Best practices:
 - Use character strings to express date and time values
 - Use language-neutral formats

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate = '20070825';
```

SQL Server can interpret a wide variety of string 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'.

This example shows the use of a string literal to extract orders with an order date of August 25, 2007:

String Literals Example

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate = '20070825';
```

Various other language-neutral formats for date and time literals 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.nnnnnnnn' 'YYYY-MM-DD hh:mm:ss.nnnnnnnn' 'YYYY-MM-DDThh:mm:ss.nnnnnnnn' '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.nnnnnnnn'	'12:30:15.1234567'
datetimeoffset	'YYYYMMDD hh:mm:ss.nnnnnnnn [+ -]hh:mm' 'YYYY-MM-DD hh:mm:ss.nnnnnnnn [+ -]hh:mm' 'YYYYMMDD' 'YYYY-MM-DD'	'20140212 12:30:15.1234567 +02:00' '2014-02-12 12:30:15.1234567 +02:00' '20140212' '2014-02-12'

Working Separately with Date and Time

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. The **datetimeoffset** data type 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 should be aware of how they behave when provided with only partial data:

- If only the date is provided, the time portion of the data type is filled with zeros and the time is considered to be set at midnight.

The following example demonstrates the behavior of **datetime2** when only date information is provided:

datetime2 with No Time

```
DECLARE @DateOnly AS datetime2 = '20160112';
SELECT @DateOnly AS Result;
```

Returns:

```
Result
-----
2016-01-12 00:00:00.0000000
```

- If a data type that holds both date and time—such as **datetime** or **datetime2**—data is populated only with time data, the date portion of the value will be set to a default value of January 1, 1900. If you need to store time data alone, use the **time** data type.

The following example shows the default date being used when time-only data is converted to the **datetime2** data type:

Default Date Example

```
DECLARE @time AS time = '12:34:56';
SELECT CAST(@time AS datetime2) AS Result;
```

Returns:

```
Result
-----
1900-01-01 12:34:56.0000000
```

- **datetime**, **smalldatetime**, **datetime2**, and **datetimeoffset** include both date and time data
- If only date is specified, time set to midnight (all zeros)

```
DECLARE @DateOnly AS datetime2 = '20160112';
SELECT @DateOnly AS Result;
```

- If only time is specified, date set to base date (January 1, 1900)

```
DECLARE @time AS time = '12:34:56';
SELECT CAST(@time AS datetime2) AS Result;
```

Querying Date and Time Values

When querying date and time data types, you might need to consider both the date and time portions of the data to return the results you expect.

In this example, a user is trying to query all the sales orders with an order date of August 25, 2007:

- Date values converted from character literals often omit time
- Queries written with equality operator for date will match midnight

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate= '20070825';
```

- If time values are stored, queries need to account for time past midnight on a date
- Use range filters instead of equality

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20070825'
AND orderdate < '20070826';
```

Midnight Time Values Example

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate= '20070825';
```

This query might satisfy the user's requirement—note that only sales orders with an order date of midnight on August 25, 2007 are returned:

orderid	custid	empid	orderdate
10643	1	6	2007-08-25 00:00:00.000
10644	88	3	2007-08-25 00:00:00.000

This is because SQL Server implicitly converts the string literal '20070825' used in the query to the same data type as the Sales.Orders.orderdate column—**datetime**—and in doing so applies the default value of midnight for the time portion of the value.

This means that the query is interpreted as:

Midnight Time Values Example (2)

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate= '20070825 00:00:00.000';
```

This means that only values that exactly match midnight are returned. If there are rows in the table with an order date of August 25, 2007 but with a time after midnight, they would not be returned by this query.

One way to be certain of returning all the orders for August 25, 2007—regardless of the time portion of the orderdate column—would be to query the data with a range, rather than a single value:

Querying a Date Range Example

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20070825'
AND orderdate < '20070826';
```

Date and Time Functions

SQL Server provides a number of functions designed to manipulate date and time data:

- Functions that return current date and time, offering you choices between various return types, in addition to 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. This offers an alternative to providing dates as string literals, as already covered in this lesson. Note that these functions require all parts of the target date/time data to be provided.
- Functions that modify date and time values, including to increment dates, to calculate the last day of a month, and to alter time zone offset information.
- Functions that examine date and time values, returning metadata or calculations about intervals between input dates.

- To get system date and time values
 - For example, GETDATE, GETUTCDATE, SYSDATETIME
- To get date and time parts
 - For example, DATENAME, DATEPART
- To get date and time values from their parts
 - For example, DATETIME2FROMPARTS, DATEFROMPARTS
- To get date and time difference
 - For example, DATEDIFF, DATEDIFF_BIG
- To modify date and time values
 - For example, DATEADD, EOMONTH
- To validate date and time values
 - For example, ISDATE

For details of all date and time functions, see the SQL Server 2016 Technical Documentation at:

 **Date and Time Data Types and Functions (Transact-SQL)**

<http://aka.ms/ifob87>

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. In Solution Explorer, open the **31 - Demonstration C.sql** script file.
2. In the **Available Databases** list, click **AdventureWorksLT**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. Select the code under the comment **Step 6**, and then click **Execute**.
8. Select the code under the comment **Step 7**, and then click **Execute**.
9. Close SQL Server Management Studio without saving any files.

Categorize Activity

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

Items	
1	datetime
2	datetime2
3	DATEFROMPARTS
4	smalldatetime
5	date
6	EOMONTH
7	time
8	datetimeoffset

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

Lab: Working with SQL Server 2016 Data Types

Scenario

You are an Adventure Works business analyst who will be writing reports using corporate databases stored in SQL Server 2016. You have been given 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 and convert character, and date and time data into various formats.

Objectives

After completing this lab, you will be able to:

- Write queries that return date and time data.
- Write queries that use date and time functions.
- Write queries that return character data.
- Write queries that use character functions.

Estimated Time: 90 Minutes

Virtual machine: **20761B-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 should 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 Information About the Current Date
3. Write a SELECT Statement to Return the Date Data Type
4. Write a SELECT Statement That Uses Different Date and Time Functions
5. Write a SELECT Statement to Show Whether a Table of Strings Can Be Used as Dates

► Task 1: Prepare the Lab Environment

1. Ensure that the 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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 Information About the Current Date

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 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 Date Data Type**

- Write December 11, 2015 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.
 - The number of days between the current date and the first column (threemonths). Use the alias diffdays.
 - The number of weeks between April 4, 1992, and September 16, 2011. Use the alias diffweeks.
 - The 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: Write a SELECT Statement to Show Whether a Table of Strings Can Be Used as Dates**

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, return a NULL.
4. Execute the written statement and compare the results achieved with the desired results shown in the file D:\Labfiles\Lab06\Solution\54 - Lab Exercise 1 - Task 4 Result.txt.

MCT USE ONLY. STUDENT USE PROHIBITED

Answer the following questions:

- What is the difference between the SYSDATETIME and CURRENT_TIMESTAMP functions?
- 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.

Exercise 2: Writing Queries That Use Date and Time Functions

Scenario

The sales department wants 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 should 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 Customers with Orders in a Given Month
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 Customers with Orders in a Given Month

1. In Solution Explorer, open the T-SQL script **61 - Lab Exercise 2.sql**.
2. Write a SELECT statement to retrieve distinct values for the custid column from the Sales.Orders table. Filter the results to include only orders placed in February 2008.
3. Execute the written statement and compare your results with the desired results shown in the file 62 - Lab Exercise 2 - Task 1 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 your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\63 - Lab Exercise 2 - Task 2 Result.txt. The results will differ because they rely on the current date.

► 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 your results 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 your results 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 to the Concatenated String Which Might Contain NULL
3. Write a SELECT Statement to Retrieve Customer Contacts Based on the First Character in the Contact Name

► **Task 1: Write a SELECT Statement to Concatenate Two Columns**

1. Open the T-SQL script **71 - Lab Exercise 3.sql**, and 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 your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

► **Task 2: Add an Additional Column to the Concatenated String Which Might Contain NULL**

1. Copy the T-SQL statement in task 1 and modify it to extend the calculated column with new information from the region column. For concatenation purposes, treat a NULL in the region column as an empty string. When the region is NULL, the modified column should look like this:

Allen, Michael (city: Berlin, region:)

When the region is not NULL, the modified column should look like this:

Richardson, Shawn (city: Sao Paulo, region: SP)

2. Execute the written statement and compare your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

► **Task 3: Write a SELECT Statement to Retrieve Customer Contacts 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 your results 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 want to address customers by their first and last names. In the Sales.Customers table, there is only one column named contactname—it 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. Write a Query to Retrieve the Contact's First Name Using SUBSTRING
3. Write a SELECT Statement to Format the Customer ID
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 T-SQL script **81 - Lab Exercise 4.sql**, and 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 your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.

► **Task 2: Write a Query to Retrieve the Contact's First Name Using SUBSTRING**

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 your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\83 - Lab Exercise 4 - Task 2 Result.txt.

MCT USE ONLY STUDENT USE PROHIBITED

► **Task 3: Write a SELECT Statement to Format the Customer ID**

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 your results 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 your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\85 - Lab Exercise 4 - Task 4 Result.txt.
3. Close SQL Server Management Studio without saving any files.

Results: After this exercise, you should have an understanding of how to use the character functions.

Module Review and Takeaways

In this module, you have learned how to:

- Describe SQL Server data types, type precedence, and type conversions.
- Write queries using numeric data types.
- Write queries using character data types.
- Write queries using date and time data types.

Review Question(s)

Question: Will SQL Server be able to successfully and implicitly convert an **int** data type to a **varchar**?

Question: What data type is suitable for storing Boolean flag information, such as TRUE or FALSE?

Question: What logical operators are useful for retrieving ranges of date and time values?

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	7-8
Lesson 3: Generating Automatic Column Values	7-12
Lab: Using DML to Modify Data	7-15
Module Review and Takeaways	7-18

Module Overview

Transact-SQL (T-SQL) data manipulation language (DML) is the subset of the SQL Language that contains commands to add and modify data column values, within rows, within tables. In this module, you will learn the basics of using INSERT to add column values to rows within tables, using UPDATE to make changes to column values to rows within tables, and using DELETE to remove complete rows from tables. You can also use the TRUNCATE command to delete all rows within a table quickly, without incurring an overhead that protects accidental deletion of rows when using the DELETE statement.

You will also learn how to generate sequences of numbers using the IDENTITY property of a column, in addition to the sequence object, which is a stand-alone object that can be applied to many columns—in the same or different tables—to gain consistency between identities within different tables.

You can use the MERGE command to change existing columns within rows of a destination table, based on the values stored within a source table, and comparisons between the source and destination table contents.

Objectives

After completing this module, you will be able to:

- Write T-SQL statements that insert column values into rows within the tables.
- Write T-SQL statements that modify values in columns, within rows, within tables.
- Write T-SQL statements that remove existing rows from tables.
- Appreciate the importance of the WHERE clause when using data modification language (DML).
- Appreciate T-SQL statements that automatically generate values for columns and see how this affects you when using DML.
- Understand the use of the MERGE statement to compare and contrast two tables and direct different DML statements, based on their content comparisons.

Lesson 1

Adding Data to Tables

In this lesson, you will learn how to write queries that add new rows with column values 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 without resort to data definition language (DDL).
- Describe the behavior of default constraints when rows are inserted into a table.

Using INSERT to Add Data

In SQL, the INSERT statement is used to add one or more rows to a table. There are several forms of the statement.

Its basic syntax appears below:

The INSERT ... VALUES statement inserts a new row

```
INSERT INTO Sales.OrderDetails  
(orderid, productid, unitprice, qty, discount)  
VALUES (10255, 39, 18, 2, 0.05);
```

Table and row constructors add multirow capability to INSERT ... VALUES

```
INSERT INTO Sales.OrderDetails  
(orderid, productid, unitprice, qty, discount)  
VALUES  
(10256, 39, 18, 2, 0.05),  
(10258, 39, 18, 5, 0.10);
```

INSERT Syntax

```
INSERT [INTO] <Table or View> [(column_list)] -- column_list is optional but the code is  
safer with it  
VALUES ([ColumnName or an expression or DEFAULT or NULL], ...n)
```

With this form, called INSERT VALUES, you can specify the columns that will have values placed in them and the order in which the data will be presented for each row inserted into the table. In addition, you can provide the values for those columns as a comma separated list.

When inserting values, the keyword DEFAULT means the predefined value that should be presented where a column value has not been listed, but a value is required.

When inserting values, the keyword NULL means the predefined value that should be presented where a column value has not been listed and a value is not required.

The following example shows the use of the INSERT VALUES statement:

Notice the correlation between the columns and the value list.

INSERT VALUES Example

```
USE TSQL  
GO
```

```
INSERT INTO Sales.OrderDetails (OrderID, ProductID, UnitPrice, Qty, Discount)  
VALUES (10248, 39, 18, 2, 0.05)
```

If the column list is omitted, a column value or the keyword (DEFAULT or NULL) must be specified for each column, 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 an IDENTITY column, 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 by providing multiple comma separated sets of values, themselves separated by commas, like this:

(1,2,3), (3,2,1), (2,2,2).

You can use T-SQL to insert multiple rows.

Insert Rows

```
USE TSQL
GO
INSERT INTO Sales.OrderDetails(orderid, productid, unitprice, qty, discount)
VALUES (10249,39,18,2,0.05), (12002,39,18,5,0.10);
-- Some people prefer this alternative layout for multiple row inserts
INSERT INTO Sales.OrderDetails(orderid, productid, unitprice, qty, discount)
VALUES (10250,39,18,2,0.05)
,
(10251,39,18,5,0.10)
,
(10252,39,18,2,0.05)
,
(10254,39,18,5,0.10);
```

INSERT (Transact-SQL)

<http://aka.ms/ifsc6i>

Table Value Constructor (Transact-SQL)

<http://aka.ms/rnwb93>

Using INSERT with Data Providers

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 can pass the results of a SELECT clause or the output of a stored procedure to the INSERT clause.

To use the SELECT statement with an INSERT statement, build a SELECT clause to replace the VALUES clause. With this form, called INSERT SELECT, you can 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 column values or DEFAULT, or NULL, for each column.

- INSERT ... SELECT to insert rows from another table:

```
INSERT Sales.OrderDetails
(orderid, productid, unitprice, qty, discount)
SELECT * FROM NewOrderDetails
```

- INSERT ... EXEC is used to insert the result of a stored procedure or dynamic SQL expression into an existing table:

```
INSERT INTO Production.Products
(productID, productname, supplierid, categoryid, unitprice)
EXEC Production.AddNewProducts;
```

The following syntax illustrates the use of INSERT using the SELECT clause:

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 into a Table from a Stored Procedure

```
INSERT INTO Production.Products (productID, 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

In T-SQL, you can use the SELECT INTO statement 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.

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://aka.ms/qae4zn>

SELECT INTO

```
SELECT column1
      , column2
```

...

```
INTO NewTable FROM OldTable
```

```
SELECT ordered
      , custid
      , empid
      , orderdate
      , shipcity
      , shipregion
      , shipcountry
INTO Sales.OrdersExport FROM Sales.Orders
WHERE empid = 5;
```

SELECT -> INTO is similar to INSERT <- SELECT

- It also creates a table for the output, fashioned on the output itself
- The new table is based on query column structure
 - Uses column names, data types, and null settings
 - Does not copy constraints or indexes

```
SELECT * INTO NewProducts FROM PRODUCTION.PRODUCTS
WHERE ProductID >= 70
```



Note: The use of SELECT INTO requires permissions to create table objects in the destination database. Do not try to put this clause inside a view, because it will only work once. If a table cannot be created when the view is activated, an error will occur after the first use of the view.

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 INTO <TableName> (<Columns,...>) VALUES (<Column Value> ...)
	INSERT INTO <DestinationTableName> SELECT <Columns> FROM <SourceTableName>
	INSERT INTO <DestinationTableName> EXECUTE usp_SomeStoredProcedure
	SELECT <Columns,...> INTO DestinationTableName FROM SourceTableName
	SELECT <Columns,...> INTO SourceTableName FROM DestinationTableNAme

Demonstration: Adding Data to Tables

In this demonstration, you will see how to:

- ADD data to tables using fully qualified parameters.
- ADD data to tables with partially qualified parameters.
- Understand how to use the OUTPUT clause to monitor data changes during data INSERT.
- Understand how to insert data into a table that is produced by a stored procedure.
- Use the SELECT INTO keywords to insert data into a table.

Demonstration Steps

INSERT Data into a Table

1. Start the 20761B-MIA-DC and 20761B-MIA-SQL virtual machines, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **D:\Demofiles\Mod07\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. In the Command Prompt window press **y**, and then press Enter.
5. When the script has finished, press Enter.
6. Open **SQL Server Management Studio**, and connect to the **MIA-SQL** database engine instance using Windows authentication.
7. On the **File** menu, point to **Open**, and then click **Project/Solution**.

8. In the **Open Project** dialog box, navigate to the **D:\Demofiles\Mod07\Demo** folder, click **Demo.ssmssln**, and then click **Open**.
9. In Solution Explorer, expand **Queries**, and double-click **11 - Demonstration A.sql**.
10. Highlight the code **USE TSQL GO**, and click **Execute**.
11. First you will populate a table with some data from a stored procedure. Highlight the code under the comment that begins **-- First try the INSERT by stored procedure:**

```
INSERT INTO Production.Products
(
    productID
    , productname
    , supplierid
    , categoryid
    , unitprice)
EXEC Production.AddNewProducts;
```

Click **Execute**. You will receive a message saying that the procedure is not there.

12. Highlight the code below the comment **--Create a backup of the Products with a chosen ID**, and click **Execute**.

```
DROP TABLE IF EXISTS NewProducts
GO

SELECT * INTO NewProducts
FROM PRODUCTION.PRODUCTS WHERE ProductID >= 70
```

You are creating a new table for NewProducts where the Product ID >= 70.

13. You are also going to create a NewOrderDetails table that will contain rows for those products that have been transferred into NewProducts. To do this, highlight the code under the comment **-- Create a backup of the Order Details for the chosen productID**, up to the point shown in the code section for the next step below, and click **Execute**:

```
DROP TABLE IF EXISTS NewOrderDetails
GO

SELECT * INTO NewOrderDetails
FROM SALES.OrderDetails WHERE ProductID >= 70

-- Delete the copied data from the original tables
DELETE FROM SALES.OrderDetails
OUTPUT DELETED./*
WHERE ProductID >= 70

DELETE FROM Production.Products
OUTPUT DELETED./*
WHERE ProductID >= 70

-- check that they have been transferred safely
SELECT * FROM NewProducts
SELECT * FROM NewOrderDetails

SELECT * FROM SALES.OrderDetails
WHERE productid >= 70

SELECT * FROM Production.Products
WHERE productid >= 70
```

14. Highlight the code below the comment **Now we can put back the rows from the NewTables, using the INSERT statement**, and click **Execute**.

```

DROP PROCEDURE IF EXISTS Production.AddNewProducts
GO

CREATE PROCEDURE Production.AddNewProducts
AS
BEGIN
SELECT Productid, productname, SupplierID, CategoryID, Unitprice FROM NewProducts
END

```

When you click Execute, SQL Server creates the stored procedure that you were missing when you tried to run it at the beginning of the demo.

15. Now you need to populate the original products table with the data within the secondary table as if you were adding new rows. Highlight the code below the comment **Having created it, we can run it to feed the missing rows into the Products table**:

```

INSERT INTO Production.Products (productid, productname, supplierid, categoryid,
unitprice)
EXEC Production.AddNewProducts;

SELECT * FROM Production.Products
WHERE productid >= 70

```

Click **Execute** to transfer the rows and see that they have been transferred.

16. For the other table, you will use the SELECT INSERT statement. Highlight the code below the comment **-- The OrderDetails will be put back using INSERT .. SELECT**, and click **Execute**:

```

INSERT Sales.OrderDetails (orderid, productid, unitprice, qty, discount)
OUTPUT INSERTED.*
SELECT * FROM NewOrderDetails

```

17. Having seen various ways to add data to a new or existing table, you can clean up the database by dropping the objects used in this demo. Highlight the rest of the code below **-- Clean up the database** and click **Execute**:

```

DROP TABLE NewProducts
GO

DROP TABLE NewOrderDetails
GO

DROP PROCEDURE Production.AddNewProducts

```

18. Close SQL Server Management Studio, without saving any changes.

MCT LICE ONLY STUDENT USE PROHIBITED

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 a MERGE between source and destination tables, 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, either 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.

- UPDATE changes all rows in a table or view
- Unless rows are filtered with a WHERE clause or constrained with a JOIN clause
- Column values are changed with the SET clause

```
UPDATE Production.Products
      SET unitprice = (unitprice * 1.04)
 WHERE categoryid = 1 AND discontinued = 0

UPDATE Production.Products
      SET unitprice *= 1.04
          -- Using compound
          -- assignment operators
 WHERE categoryid = 1 AND discontinued = 0;
```

 **Note:** It's important to note that an UPDATE without a corresponding WHERE clause, and/or a join, will target all rows that are not filtered out of the operation. Use the UPDATE statement with caution.

The following code shows the basic syntax of the UPDATE statement:

UPDATE Syntax

```
UPDATE <TableName>
SET
    <ColumnName1> = { expression | DEFAULT | NULL }
    { ,...n }
```

Any column omitted from the SET clause will not be modified by the UPDATE operation.

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;
```



Note: In an earlier module, you learned that T-SQL supports compound assignment operators. These can be used when assigning values to columns using the SET statement within the update clause, as shown below:

```
UPDATE Production.Products
```

```
    SET unitprice *= 1.04
```

```
    WHERE categoryID = 1
```

```
    AND discontinued = 0;
```



UPDATE (Transact-SQL)

<http://aka.ms/sbikqm>

Using MERGE to Modify Data

In database operations, there is a common need to perform a SQL MERGE operation, in which some rows within a destination table are updated or deleted and new rows are inserted from a source data table. The oldest versions of SQL Server, before support for the MERGE statement was added, required multiple operations to update and insert data into a destination table. You can use the MERGE statement to insert, update, and even delete rows from a destination table, based on a join to a source data set, all in a single statement.

MERGE modifies data, based on one or more conditions:

- When the source data matches the data in the target, it updates data.
- When the source data has no match in the target, it inserts data.
- When the target data has no match in the source, it deletes the target data.



Note: Because the T-SQL implementation of MERGE supports the WHEN NOT MATCHED BY SOURCE clause, MERGE is more than just an upsert operation—because it also deletes, it is a delupsert or something similar.

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:

The 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 TargetTbl.col2 = SourceTbl.col2
    WHEN NOT MATCHED THEN
        INSERT (<column_list>)
        VALUES (<value_list>);
```

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 TOP (10) INTO Store AS Destination
    USING StoreBackup AS StagingTable
    ON(Destination.Key = StagingTable.Key)

    WHEN NOT MATCHED THEN
        INSERT (C1,...)
        VALUES (Source.C1,...)
    WHEN MATCHED THEN
        UPDATE SET Destination.C1 = StagingTable.C1,...
```

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 top (10) INTO Store          AS Destination      -- Known in online help as
Target, which is a reserved word
    USING     StoreBackup AS StagingTable -- Known in online help as the source, which
is also a reserved word
    ON          (Destination.BusinessEntityID =
StagingTable.BusinessEntityID)                                -- the matching
control columns
WHEN NOT MATCHED THEN
    INSERT (
        ,          BusinessEntityID
        ,          Name
        ,          SalesPersonID
        ,          Demographics
        ,          rowguid
        ,          ModifiedDate
    )
    VALUES (
        ,          StagingTable.BusinessEntityID
        ,          StagingTable.Name
        ,          StagingTable.SalesPersonID
        ,          StagingTable.Demographics
        ,          StagingTable.rowguid
        ,          StagingTable.ModifiedDate
    );
;
```



MERGE (Transact-SQL)

<http://aka.ms/nbsfg7>

Demonstration: Manipulating Data Using the UPDATE and DELETE Statements and MERGING Data Using Conditional DML

In this demonstration, you will see how to:

- UPDATE row and column intersections within tables.
- DELETE complete rows from within tables.
- Apply multiple data manipulation language (DML) operations by using the MERGE statement.
- Understand how to use the OUTPUT clause to monitor data changes during DML operations.
- Understand how to access prior and current data elements, in addition to showing the DML operation performed.

Demonstration Steps

Update and Delete Data in a Table

1. Start the 20761B-MIA-DC and 20761B-MIA-SQL virtual machines, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$wOrd**.
2. Run **D:\Demofiles\Mod07\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. In the Command Prompt window press **y**, and then press Enter.

MCT USE ONLY STUDENT USE PROHIBITED

5. When the script has finished, press Enter.
6. Start **SQL Server Management Studio**, and connect to the **MIA-SQL** database engine instance using Windows authentication.
7. On the **File** menu, point to **Open**, and then click **Project/Solution**.
8. In the **Open Project** dialog box, navigate to the **D:\Demofiles\Mod07\Demo** folder, click **Demo.ssmssln**, and then click **Open**.
9. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
10. Highlight the code **USE AdventureWorks GO**, and click **Execute**.
11. Select the code under **USE AdventureWorks GO**, and then click **Execute**.
12. Select the code under the comment **Remove the copied rows from the store table**, and then click **Execute**.
13. Select the code under the comment **Show that they have been removed**, and then click **Execute**.
14. Select the code under the comment **Use the Merge statement to put them back**, and then click **Execute**.
15. Select the code under the comment **SELECT * FROM Sales.Store where 1 = 0 -- used to extract column names for all columns, without cost of data access**, and then click **Execute**.
16. Select the code under the comment **Use the Merge statement to Change the names back**, and then click **Execute**.
17. Select the code under the comment **Ensure that the environment has been restored to the state it was in before the changes were made**, and then click **Execute**.
18. Select the code under the comment **Clean up the database**, and then click **Execute**.
19. Close SQL Server Management Studio without saving any files.

Question: A user cannot delete records in the Cars table by using a DELETE statement. The query was intended to remove all pool cars that have been sold. The query used was:

```
DELETE
FROM Scheduling.Cars
WHERE Cars.DateSold <> NULL
```

What mistake did the user make?

Lesson 3

Generating Automatic Column Values

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 when rows are inserted into a table.
- Describe how to use a sequence object in SQL Server 2016 to generate numbers that can be used within a column, in one or more tables.

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 2012-2016. Each mechanism can be used to provide sequential numbers when rows are inserted into a table. With the sequence object, the number variable can be used efficiently in multiple tables.

To use the IDENTITY property, define a column using a numeric data type with a scale of 0—meaning whole numbers only—and include the IDENTITY keyword.

An optional seed (starting value), and an increment (step value) can also be specified. Leaving out the seed and increment will set them both to 1.

Only one column in a table may have the IDENTITY property set; it is customary for it to be an alternate primary key.

The following code fragment shows an EmployeeID column defined with the IDENTITY property, a seed of 100, and an increment of 10:

IDENTITY Example

```
CREATE TABLE Employee
(
    EmployeeID int IDENTITY(100, 10) NOT NULL
);
```

The IDENTITY property generates column values automatically

- Optional seed and increment values can be provided

```
CREATE TABLE Production.Products
(PID int IDENTITY(1,1) NOT NULL, Name VARCHAR(15), ...)
```

- Only one column in a table may have IDENTITY defined
- IDENTITY column must be omitted in a normal INSERT statement

```
INSERT INTO Production.Products (Name,...)
VALUES ('MOC 2072 Manual',...)
```

- Functions are provided to return last generated values
 - SELECT @IDENTITY: default scope is session
 - SELECT SCOPE_IDENTITY(): scope is object containing the call
 - SELECT IDENT_CURRENT('tablename'): in this case, scope is defined by tablename
- There is a setting to allow identity columns to be changed manually ON or automatic OFF
 - SET IDENTITY_INSERT <tablename> {ON|OFF}

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 for 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.

For more information, see SET IDENTITY_INSERT (Transact-SQL) in the SQL Server 2016 Technical Documentation:

 **SET IDENTITY_INSERT (Transact-SQL)**
<http://aka.ms/l4wvg>

When 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. You can use SCOPE_IDENTITY() for most purposes.

To reset the IDENTITY property by assigning a new seed, use the DBCC CHECKIDENT statement.

For more information, see DBCC CHECKIDENT (Transact-SQL) in the SQL Server 2016 Technical Documentation:

 **DBCC CHECKIDENT (Transact-SQL)**
<http://aka.ms/g3mejh>

Check Your Knowledge

Question	
You are using an IDENTITY column to store the sequence 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

Using Sequences

As you have learned, the IDENTITY property is 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 need to create tables of numbers manually to provide a pool of sequential values across tables.

SQL Server 2012 provides the new sequence object, an independent database object that 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.

For more information, see CREATE SEQUENCE (Transact-SQL) in the SQL Server 2016 Technical Documentation:

 **CREATE SEQUENCE (Transact-SQL)**
<http://aka.ms/lquwo6>

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.

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.tb1Demo
    (SeqCol int PRIMARY KEY,
     ItemName nvarchar(25) NOT NULL);
GO

INSERT
    INTO dbo.tb1Demo (SeqCol,ItemName)
    VALUES
        (NEXT VALUE FOR dbo.demoSequence, 'Item');
GO
```

When you use a select statement against the table, you will see that a sequence value is inserted for the new row.

Sequence objects were first 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;
```

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 template 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: **20761B-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 TempDB.Hr.Employee table and test the required T-SQL code. You can then pass the T-SQL code 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 a SELECT Statement As the Data Provider

► Task 1: Prepare the Lab Environment

1. Ensure that the 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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 execute the **01 setup.sql** query.
2. Write an **INSERT** statement to add a record to the **Employees** table within the **TempDB.HR.Employees** table, with the following values:
 - Title: Sales Representative
 - Titleofcourtesy: Mr
 - FirstName: Laurence
 - Lastname: Grider
 - Hiredate: 04/04/2013
 - Birthdate: 10/25/1975
 - Address: 1234 1st Ave. S.E.
 - City: Seattle
 - Country: USA
 - Phone: (206)555-0105

► **Task 3: Insert a Row with a SELECT Statement As the Data Provider**

- 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—making searches more straightforward. You also want to remove the three potential customers who have 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**

- Write an **UPDATE** statement to update all the records in the **Customers** table that have a city of 'Berlin' and a contacttitle of 'Sales Representative' to have a contacttitle of 'Sales Consultant'.

► **Task 2: Delete Rows**

- 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 that have a city of Berlin and a contacttitle of Sales Representative, to now 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.

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 successfully?

Module Review and Takeaways

In this module, you have learned how to:

- Write T-SQL statements that insert column values into rows within the tables.
- Write T-SQL statements that modify values in columns, within rows, within tables.
- Write T-SQL statements that remove existing rows from tables.
- Appreciate the importance of the WHERE clause when using data modification language (DML).
- Appreciate T-SQL statements that automatically generate values for columns and how this affects you when using DML.
- Understand the use of the MERGE statement to compare and contrast two tables and direct different DML statements, based on their content comparisons.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
You are partway through the exercises and want to start again from the beginning. You run the setup script within the solution and receive lots of error messages. This might occur if you have tried to execute the setup script without running the cleanup script to remove any changes you might have made during the lab.	

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	8-16
Lab: Using Built-in Functions	8-20
Module Review and Takeaways	8-24

Module Overview

In addition to retrieving data as it is stored in columns, you may have to compare or further manipulate values in your T-SQL queries.

In this module, you will:

- Learn about the many built-in functions in Microsoft® SQL Server® that provide data type conversion, comparison, and NULL handling.
- Learn about the various types of functions in SQL Server and how they are categorized.
- Work with scalar functions and see where they may be used in your queries.
- Learn conversion functions for changing data between different data types, and how to write logical tests.
- Learn how to work with NULLs, and use built-in functions to select non-NULL values, in addition to replacing 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 SQL Server function types, and then work 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:

• SQL Server functions can be categorized by scope of input and type of output:	
Function Category	Description
Scalar	Operate on a single row, return a single value
Grouped Aggregate	Take one or more values but return a single summarizing value
Window	Operate on a window (set) of rows
Rowset	Return a virtual table that can be used in a T-SQL statement

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 in a T-SQL statement



Note:

- This course will cover aggregates and window functions in later modules.
- Rowset functions are beyond the scope of this course.
- The rest of this module will cover various scalar functions.

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). As scalar functions always return a single value, they can 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 nondeterministic, 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; others use the collation of the database if no input collation is supplied.

At the time of writing, the SQL Server 2016 Technical Documentation listed more than 200 scalar functions. This course is not intended to provide a complete guide to all functions. The following list provides some representative examples:

- Date and time functions (covered previously in this course).
- Mathematical functions.
- Conversion functions (covered later in this module).
- System metadata functions.
- System functions.
- Text and image 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

- Operate on elements from a single row as inputs, return a single value as output
- Return a single (scalar) value
- Can be used like an expression in queries
- May be deterministic or nondeterministic
- Collation depends on input value or default collation of database

Scalar Function Categories

- Configuration
- Conversion
- Cursor
- Date and Time
- Logical
- Mathematical
- Metadata
- Security
- String
- System
- System Statistical
- Text and Image

NOTICE 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
```

For additional information about scalar functions and categories, see the SQL Server 2016 Technical Documentation:

Built-in Functions (Transact SQL)

<http://aka.ms/oor5qi>

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; aggregation is performed on all of them.

The following example uses a COUNT function and a SUM function to return aggregate values without a GROUP BY clause:

Aggregate Function

```
SELECT COUNT(*) AS numorders, SUM(unitprice) AS totalsales  
FROM Sales.OrderDetails;
```

The Results:

```
numorders totalsales  
----- -----  
2155 56500.91
```

- 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
```



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, 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:

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 QHFFO	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 enables 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.

Further discussion of rowset functions is beyond the scope of this course. For more information, see the SQL Server 2016 Technical Documentation:

 **Rowset Functions (Transact-SQL)**

<http://go.microsoft.com/fwlink/?LinkId=402746>

- 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
38	Product QDOMO	263.50	1
29	Product VJXYN	123.79	2
9	Product AOZBW	97.00	3
20	Product QHFFO	81.00	4
18	Product CKEDC	62.50	5

- 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

Demonstration: Writing Queries Using Built-in Functions

In this demonstration you will see how to:

- Use build-in scalar functions.

Demonstration Steps

Use Built-in Scalar Functions

1. Ensure that the 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **D:\Demofiles\Mod08\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. At the command prompt, type **y**, and then press Enter.
5. When the script has finished, press Enter.
6. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
7. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod08\Demo** folder.
8. In Solution Explorer, expand **Queries**, and then double-click **11 - Demonstration A.sql**.
9. Select the code under the comment **Step 1**, and then click **Execute**.
10. Select the code under the comment **Step 2**, and then click **Execute**.
11. Select the code under the comment **Step 3**, and then click **Execute**.
12. Select the code under the comment **Step 4**, and then click **Execute**.
13. Keep SQL Server Management Studio open for the next demonstration.

MCT USE ONLY. STUDENT USE PROHIBITED

Categorize Activity

Categorize each item into the appropriate category. Indicate your answer by writing the category number to the right of each item.

Items	
1	GETDATE()
2	SUM()
3	OPENDATASOURCE()
4	DATEADD()
5	MIN()
6	OPENQUERY()
7	UPPER()
8	MAX()
9	OPENROWSET()
10	YEAR()
11	COUNT()
12	OPENXML()
13	ABS()
14	AVG()
15	DB_NAME()

Category 1	Category 2	Category 3
Scalar Functions	Aggregate Functions	Rowset Functions

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; 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 2016 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 might need to override the type precedence, or force a conversion where an implicit conversion might fail.

To accomplish this, you can use the CAST and CONVERT functions, in addition to 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 noncharacter 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. The end result 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.

For additional reading about truncation behavior, see the SQL Server 2016 Technical Documentation:



<http://go.microsoft.com/fwlink/?LinkId=402747>

- Implicit conversion occurs automatically and 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

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 following example shows how to use the CAST function:

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 CATS, see the SQL Server 2016 Technical Documentation:

 **CAST and CONVERT (Transact-SQL)**

<http://go.microsoft.com/fwlink/?LinkId=402747>

MERGE 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 occasionally still need to use CONVERT.

As with CAST, CONVERT is a scalar function. You may use CONVERT in SELECT and WHERE clauses.

The following example shows how to use the CONVERT function:

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, in addition to 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 yyyyymmdd

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, see the SQL Server 2016 Technical Documentation:

CAST and CONVERT (Transact-SQL)

<http://go.microsoft.com/fwlink/?LinkId=402747>

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 2016 makes this task easier with the PARSE function.

PARSE requires 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 converts strings to date, time, and number types:

PARSE element	Comment
String_value	Formatted nvarchar(4000) input
Data_type	Requested data type output
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 with Culture Code

```
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 the SQL Server 2016 Technical Documentation:

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:

- 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 2016 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:

```
-----  
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. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**. Note the error message.
5. Select the code under the comment **Step 4a**, and then click **Execute**.
6. Select the code under the comment **Step 4b**, and then click **Execute**. Note the error message.
7. Select the code under the comment **Step 5**, and then click **Execute**.
8. Select the code under the comment **Step 6**, and then click **Execute**.
9. Select the code under the comment **Step 7**, and then click **Execute**.
10. Select the code under the comment **Step 8a**, and then click **Execute**. Note the error message.
11. Select the code under the comment **Step 8b**, and then click **Execute**.
12. 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?

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 Test 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:

- ISNUMERIC tests whether an input expression is a valid numeric data type:

- Returns a 1 when the input evaluates to any valid numeric type, including FLOAT and MONEY
- Returns 0 otherwise

- Example:

```
SELECT ISNUMERIC('SQL') AS isnumeric_result;
isnumeric_result
0
SELECT ISNUMERIC('101.99') AS isnumeric_result;
isnumeric_result
1
```

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?

MCT USE ONLY. STUDENT LICENSE PROHIBITED

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 VBA, Excel and Access, 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:								
<table border="1"> <tr> <td>IIF Element</td> <td>Comments</td> </tr> <tr> <td>Boolean_expression</td> <td>Logical test evaluating to TRUE, FALSE, or UNKNOWN</td> </tr> <tr> <td>True_value</td> <td>Value returned if expression evaluates to TRUE</td> </tr> <tr> <td>False_value</td> <td>Value returned if expression evaluates to FALSE or UNKNOWN</td> </tr> </table>	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
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							
• IIF example:								
<pre>SELECT productid, unitprice, IIF(unitprice > 50, 'high','low') AS pricepoint FROM Production.Products;</pre>								

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, see *IIF (Transact-SQL)* in the SQL Server 2016 Technical Documentation:



IIF (Transact-SQL)

<http://go.microsoft.com/fwlink/?LinkId=402748>

Selecting Items from a List with CHOOSE

CHOOSE returns the value of an item at a specific index in a list.

CHOOSE returns an item from a list, selecting the item that matches an index value:

- CHOOSE returns an item from a list as specified by an index value:

CHOOSE Element	Comments
Index	Integer that represents position in list
Value_list	List of values of any data type to be returned

- CHOOSE example:

```
SELECT CHOOSE (3, 'Beverages', 'Condiments', 'Confections')
AS choose_result;
```

```
choose_result
-----
Confections
```

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.

CHOOSE (Transact-SQL)

<http://aka.ms/kt4v4m>

Demonstration: Using Logical Functions

In this demonstration, you will see how to:

- Use logical functions.

Demonstration Steps

Using Logical Functions

1. In Solution Explorer, open the **31 - Demonstration C.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. 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?

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																							
• ISNULL example:																								
SELECT custid, city, ISNULL(region, 'N/A') AS region, country FROM Sales.Customers;																								
<table border="1"> <thead> <tr> <th>custid</th> <th>city</th> <th>region</th> <th>country</th> </tr> </thead> <tbody> <tr> <td>7</td> <td>Strasbourg</td> <td>N/A</td> <td>France</td> </tr> <tr> <td>9</td> <td>Marseille</td> <td>N/A</td> <td>France</td> </tr> <tr> <td>32</td> <td>Eugene</td> <td>OR</td> <td>USA</td> </tr> <tr> <td>43</td> <td>Walla Walla</td> <td>WA</td> <td>USA</td> </tr> <tr> <td>45</td> <td>San Francisco</td> <td>CA</td> <td>USA</td> </tr> </tbody> </table>	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
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.



ISNULL (Transact-SQL)

<http://go.microsoft.com/fwlink/?LinkId=402750>

Using COALESCE to Return Non-NULL Values

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 multipart 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

Code Example Content

```
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, see the SQL Server 2016 Technical Documentation:

 **COALESCE (Transact-SQL)**

<http://go.microsoft.com/fwlink/?LinkId=402751>

Using NULLIF to Return NULL If Values Match

In this module, the NULLIF function is the first you will learn 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's 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, see *NULLIF (Transact-SQL)* in the SQL Server 2016 Technical Documentation:



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

- In Solution Explorer, open the **41 - Demonstration D.sql** script file.
- Select the code under the comment **Step 1**, and then click **Execute**.
- Select the code under the comment **Step 2**, and then click **Execute**.
- Select the code under the comment **Step 3**, and then click **Execute**.

• NULLIF compares two expressions:
• Returns NULL if both arguments are equal
• Returns the first argument if the two arguments are not equal
emp_id
1
2
3
4
goal
100
90
100
100
actual
110
90
90
80
SELECT emp_id, NULLIF(goal,actual) AS actual_if_different
FROM dbo.employee_goals;
emp_id
1
2
3
4
actual_if_different
110
NULL
90
80

5. Select the code under the comment **Step 4a**, and then click **Execute**.
6. Select the code under the comment **Step 4b**, and then click **Execute**.
7. Select the code under the comment **Step 4c**, and then click **Execute**.
8. Select the code under the comment **Step 4d**, and then click **Execute**.
9. Select the code under the comment **Step 5**, and then click **Execute**.
10. 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()
	COALESCE()
	NULLIF()
	TRY_PARSE()
	PARSE()

Lab: Using Built-in Functions

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 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: **20761B-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa\$\$w0rd**

Exercise 1: Writing Queries That Use Conversion Functions

Scenario

You have been asked to write the following reports for these departments:

1. **Sales.** The product name and unit price for each product within an easy to read string.
2. **Marketing.** The order id, order date, shipping date, and shipping region for each order after 4/1/2007.
3. **IT.** Convert all Sales phone number information into integers.

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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"**).
2. 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"**.
3. In this SELECT statement, you can use the CONVERT function with a style parameter or the PARSE function.
4. 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: After this exercise, you should be able to use conversion functions.

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.
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

- 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.

Module Review and Takeaways

In this module, you have learned how to:

- Write queries with built-in scalar functions.
- Use conversion functions.
- Use logical functions.
- Use functions that work with NULL.



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?

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	9-15
Lab: Grouping and Aggregating Data	9-18
Module Review and Takeaways	9-24

Module Overview

In addition to row-at-a-time queries, you may need to summarize data to analyze it. Microsoft® SQL Server® provides 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 simultaneously.

Many aggregate functions are 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, you need to consider the following:

- 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, see the SQL Server 2016 Technical Documentation:



Aggregate Functions (Transact-SQL)

<http://aka.ms/wq6lku>

- 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

Built-in Aggregate Functions

SQL Server provides many built-in aggregate functions. Commonly used functions include:

Common	Statistical	Other
<ul style="list-style-type: none"> SUM MIN MAX AVG COUNT COUNT_BIG 	<ul style="list-style-type: none"> STDEV STDEVP VAR VARP 	<ul style="list-style-type: none"> CHECKSUM_AGG GROUPING GROUPING_ID

This lesson will only cover common aggregate functions. For more information on other built-in aggregate functions, see the SQL Server 2016 Technical Documentation.

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 largest number, earliest date/time, or first-occurring string (according to collation sort rules).
MAX	MAX(<expression>)	Returns the largest number, latest date/time, or last-occurring string (according to collation sort rules).
COUNT or COUNT_BIG	COUNT(*) or COUNT(<expression>)	With (*), counts all rows, including those with NULL values. When a column is specified as <expression>, returns count of non-NULL rows for that column. COUNT returns an int; COUNT_BIG returns a big_int.

This lesson only covers common aggregate functions. For information on other built-in aggregate functions, see the SQL Server 2016 Technical Documentation:

Aggregate Functions (Transact-SQL)

<http://aka.ms/wq6lku>

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;
```

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 query 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. Therefore, all columns 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 ZRNE

Other functions may coexist with aggregate functions.

For example, the YEAR scalar function is used in the following illustration 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 summarizing unique occurrences of values, such as customers in the TSQL orders table.

The following example returns customers who have placed orders, grouped by employee ID and year:

- 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
2	2007	41	35
2	2008	39	34
3	2006	18	16
3	2007	71	46
3	2008	38	30

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 as 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:

- 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, 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;
```

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.

- 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;
```

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 versus 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. You will find a script to create the table 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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **D:\Demofiles\Mod09\Setup.cmd** as an administrator.

3. In the **User Account Control** dialog box, click **Yes**.
4. Start SQL Server Management Studio and connect to the **MIA-SQL** database instance using Windows authentication.
5. On the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, navigate to the **D:\Demofiles\Mod09\Demo** folder, click **Demo.ssmssln**, and then click **Open**.
7. In Solution Explorer, expand **Queries**, and then double-click **11 - Demonstration A.sql**.
8. Select the code under the comment **Step 1**, and then click **Execute**.
9. Select the code under the comment **Step 2a**, and then click **Execute**.
10. Select the code under the comment **Step 2b**, and then click **Execute**.
11. Select the code under the comment **Step 2c**, and then click **Execute**.
12. Select the code under the comment **Step 2d**, and then click **Execute**.
13. Select the code under the comment **Step 2e**, and then click **Execute**.
14. Select the code under the comment **Step 2f**, and then click **Execute**.
15. Select the code under the comment **Step 2g**, and then click **Execute**.
16. Select the code under the comment **Step 3a**, and then click **Execute**.
17. Select the code under the comment **Step 3b**, and then click **Execute**.
18. Select the code under the comment **Step 3c**, and then click **Execute**.
19. Select the code under the comment **Step 3d**, and then click **Execute**.
20. Select the code under the comment **Step 3e**, and then click **Execute**.
21. Select the code under the comment **Step 3f**, and then click **Execute**.
22. Select the code under the comment **Step 3g**, and then click **Execute**.
23. Select the code under the comment **Step 3h**, and then click **Execute**.
24. Select the code under the comment **Step 3i**, and then click **Execute**.
25. Select the code under the comment **Step 4**, and then click **Execute**.
26. 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?

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 is created. The contents of the virtual table are now available for further processing. You can use the GROUP BY clause to subdivide the results of the preceding query phases into groups of rows.

To group rows, specify one or more elements in the GROUP BY clause:

- 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 the GROUP BY clause is processed

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 SalesOrders
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 the results of the GROUP BY clause, you will need to add a SELECT clause.

MCT USE ONLY. STUDENT USE PROHIBITED

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, see GROUP BY (Transact-SQL) in the SQL Server 2016 Technical Documentation:



GROUP BY (Transact-SQL)

<http://aka.ms/ro266s>

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.

A review of the logical order of operations during query processing will help clarify this issue.

As mentioned earlier in the course, the SELECT clause is not processed until after the FROM, WHERE, GROUP BY, and HAVING clauses (if present) are processed. 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.

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 the GROUP BY clause or be inputs to aggregate expressions

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 because 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 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 cannot combine a view of summary data with the detailed source data, using the T-SQL tools you have learned about so far. You will learn some approaches to solving the problem later in this course.

For more information about troubleshooting GROUP BY errors, see:



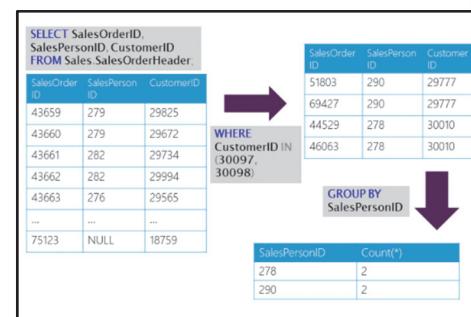
Troubleshooting GROUP BY Errors

<http://aka.ms/yi931j>

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 SalesOrderID, SalesPersonID, CustomerID
FROM Sales.SalesOrderHeader;

SELECT SalesOrderID, SalesPersonID, CustomerID
FROM Sales.SalesOrderHeader
WHERE CustomerID IN (29777, 30010);

SELECT SalesPersonID, COUNT(*)
FROM Sales.SalesOrderHeader
WHERE CustomerID IN (29777, 30010)
GROUP BY SalesPersonID;
```

Using GROUP BY with Aggregate Functions

As you have seen, 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.

- Aggregate functions are commonly used in SELECT clause, summarize per group:

```
SELECT custid, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY custid;
```

- Aggregate functions may refer to any columns, not just those in GROUP BY clause

```
SELECT productid, MAX(qty) AS largest_order
FROM Sales.OrderDetails
GROUP BY productid;
```

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
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. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2a**, and then click **Execute**.
4. Select the code under the comment **Step 2b**, and then click **Execute**.
5. Select the code under the comment **Step 3**, and then click **Execute**.
6. Select the code under the comment **Step 4a**, and then click **Execute**.
7. Select the code under the comment **Step 4b**, and then click **Execute**.
8. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge**Question**

You are writing the following T-SQL query to find out how many employees work in each department in your organization:

```
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
```

Which columns should be included in the GROUP BY clause?

Select the correct answer.

	All Columns
	EmployeeCount
	DepartmentID, DepartmentName
	DepartmentID

Lesson 3

Filtering Groups with HAVING

When you have created groups with a GROUP BY clause, you can 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:

- 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;
```

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	

A HAVING clause enables you to create a search condition, conceptually similar to the predicate of a WHERE clause, which then tests each group returned by the GROUP BY clause.

The following example from the TSQL database groups all orders by customer, then returns only those 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;
```

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.



HAVING (Transact-SQL)

<http://aka.ms/wsrrp0>

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.

- Using a COUNT(*) expression in a HAVING clause is useful to solve common business problems:

- Show only customers who 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;
```

You will see a comparison between WHERE and HAVING in the next demonstration.

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. In Solution Explorer, open the **31 - Demonstration C.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2a**, and then click **Execute**.
4. Select the code under the comment **Step 2b**, and then click **Execute**.
5. Select the code under the comment **Step 2c**, and then click **Execute**. Note the error message.
6. Select the code under the comment **Step 2d**, and then click **Execute**.
7. Select the code under the comment **Step 2e**, and then click **Execute**.
8. Select the code under the comment **Step 2f**, and then click **Execute**.
9. Select the code under the comment **Step 2g**, and then click **Execute**.
10. Select the code under the comment **Step 2h**, and then click **Execute**.
11. Select the code under the comment **Step 2i**, and then click **Execute**.
12. 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?

Lab: Grouping and Aggregating Data

Scenario

You are an Adventure Works business analyst, who will be writing reports using corporate databases stored in SQL Server. You have been given 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: **20761B-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa\$\$w0rd**

Exercise 1: Writing Queries That Use the GROUP BY Clause

Scenario

The sales department want 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 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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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 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 2 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 3 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 4 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 5 Result.txt.

Results: After this exercise, you should be able to use the GROUP BY clause in the T-SQL statement.

NO USE ONLY STUDENT USE PROHIBITED

Exercise 2: Writing Queries That Use Aggregate Functions

Scenario

The marketing department wants to launch a new campaign, so the staff need to gain a better insight into the existing customers' buying behavior. You should 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
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?

Exercise 3: Writing Queries That Use Distinct Aggregate Functions

Scenario

The marketing department want to have some additional reports that display the number of customers who made any order in a specific time period 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. Amend 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.

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. 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 to 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.
3. Close SQL Server Management Studio without saving any files.

Results: After this exercise, you should have an understanding of how to use the HAVING clause.

Module Review and Takeaways

In this lesson, you have learned how 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.

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: In a query, can a WHERE clause and a HAVING clause filter on the same column?

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	10-10
Lab: Using Subqueries	10-13
Module Review and Takeaways	10-18

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. You can also use Microsoft® SQL Server® 2016 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 suggests, 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.

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 stand-alone 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.

- 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



Note: You will learn about correlated subqueries later in this module.

For additional reading about subqueries, see the SQL Server 2016 Technical Documentation:



Subquery Fundamentals

<http://aka.ms/f6uu08>

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.

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 shown 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
```

You will then 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

- 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

NOTICE ONLY STUDENTS USE PROHIBITED

Test the logic of your subquery to ensure it will only return a single value. In the query above, because the outer query used an = operator in the predicate of the WHERE clause, and the subquery returned a single value, the query ran correctly. If an outer query is written to expect a single value, such as by using simple equality operators (=, <, >, and <>, for example), and the inner query returns more than one result, 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.

See *PRIMARY KEY Constraints* in the SQL Server 2016 Technical Documentation:



PRIMARY KEY Constraints

<http://aka.ms/acq0rx>

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);
```

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. For more information about execution plans and query performance, see Microsoft Course 20472-3: *Performance Tuning and Optimizing Microsoft SQL Server Databases*.

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. In the **D:\Demofiles\Mod10** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and wait for the script to finish.
4. At the command prompt, press any key.
5. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows® authentication.
6. On the **File** menu, point to **Open**, and then click **File**.
7. In the **Open File** dialog box, navigate to the **D:\Demofiles\Mod10\Demo** folder, and then double-click **Demo.ssmssln**.

8. In Solution Explorer, expand **Queries**, and then double-click **11 - Demonstration A.sql**.
9. Select the code under the comment **Step 1**, and then click **Execute**.
10. Select the code under the comment **Step 2**, and then click **Execute**.
11. Select the code under the comment **Step 3**, and then click **Execute**.
12. Select the code under the comment **Step 4**, and then click **Execute**. Note the error message.
13. Select the code under the comment **Step 5**, and then click **Execute**.
14. Select the code under the comment **Step 6**, and then click **Execute**.
15. 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?

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, in addition to rewriting 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.

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;

```

- 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;

```

NCT USE ONLY. STUDENT USE PROHIBITED

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, because 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?

Writing Correlated Subqueries

To write correlated subqueries, consider the following guidelines:

- 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 might return multiple values, use an IN predicate. Plan to handle NULL results.

- 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.
```

- 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.

For additional reading about correlated subqueries, see the SQL Server 2016 Technical Documentation:

Correlated Subqueries

<http://aka.ms/hoxorm>

Demonstration: Writing Correlated Subqueries

In this demonstration, you will see how to:

- Write a correlated subquery.

Demonstration Steps

Write a Correlated Subquery

- In Solution Explorer, open the **21 - Demonstration B.sql** script file.
- Select the code under the comment **Step 1**, and then click **Execute**.
- Select the code under the comment **Step 2**, and then click **Execute**.
- Select the code under the comment **Step 3**, and then click **Execute**.
- Select the code under the comment **Step 4**, and then click **Execute**.
- Keep SQL Server Management Studio open for the next demonstration.

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.

ACT 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 how it does elsewhere 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.

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, lastname
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);
```

- 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)`

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.

• The keyword EXISTS does not follow a column name or other expression

• The SELECT list of a subquery introduced by EXISTS typically only uses an asterisk (*)

```
SELECT custid, companyname
FROM Sales.Customers AS c
WHERE EXISTS (
    SELECT *
    FROM Sales.Orders AS o
    WHERE c.custid=o.custid);

SELECT custid, companyname
FROM Sales.Customers AS c
WHERE NOT EXISTS (
    SELECT *
    FROM Sales.Orders AS o
    WHERE c.custid=o.custid);
```

See *Subqueries with EXISTS* in the SQL Server 2016 Technical Documentation:

 **Subqueries with EXISTS**

<http://aka.ms/q812le>

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. In Solution Explorer, open the **31 - Demonstration C.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4a**, and then click **Execute**.
6. Select the code under the comment **Step 4b**, and then click **Execute**.
7. 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 added their skills to the database by using a web-based user interface. You want to find employees who have not yet added their skills. 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

As a business analyst for Adventure Works, you are writing reports using corporate databases stored in SQL Server. You have been handed a set of business requirements for data and 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 multiresult set subqueries.
- Write queries that use correlated subqueries and the EXISTS predicate.

Estimated Time: 60 minutes

Virtual machine: **20761B-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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. 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**.

► **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 Multiresult 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 multiresult subqueries in T-SQL statements.

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**. Make sure 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 who 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.

► **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

In this module, you have learned how 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.

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?

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® 2016 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 a 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, in addition to extra security beyond that 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 a 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.

 **Note:** In an indexed view, data is materialized in the view. Indexed views are beyond the scope of this course.

- 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>;
```

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 that 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;
```

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. Providing 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 course 20764B: *Administering a SQL Database Infrastructure*.

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.

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;
```

- 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;
```



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 course 20762B: *Developing Microsoft SQL Server Databases*.

NO 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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **D:\Demofiles\Mod11\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. At the command prompt, type **y**, press Enter, and then wait for the script to finish.
5. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
6. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod11\Demo** folder.
7. Open the **11 - Demonstration A.sql** script file.
8. Select the code under the comment **Step 1**, and then click **Execute**.
9. Select the code under the comment **Step 2**, and then click **Execute**.
10. Select the code under the comment **Step 3**, and then click **Execute**.
11. Select the code under the comment **Step 4**, and then click **Execute**.
12. 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 set up this access?

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 course 20762B: *Developing Microsoft SQL Server 2016 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 the SQL Server 2016 Technical Documentation:

 **CREATE FUNCTION (Transact-SQL)**

<http://go.microsoft.com/fwlink/?LinkId=402772>

MCT USE ONLY. STUDENT USE PROHIBITED

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.

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);
```

- 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;
```

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:** You use a two-part name 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. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Keep SQL Server Management Studio open for the next demonstration.

NOTICE 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 store data in the database but queries the database whenever it is called.

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>
```

- Derived tables are named query expressions created within an outer SELECT statement
- Not stored in database—represents a virtual relational table
- When processed, unpacked into query against underlying referenced objects
- Allow you to write more modular queries

```
SELECT <column_list>
FROM (
    <derived_table_definition>
) AS <derived_table_alias>;
```

- Scope of a derived table is the query in which it is defined

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)

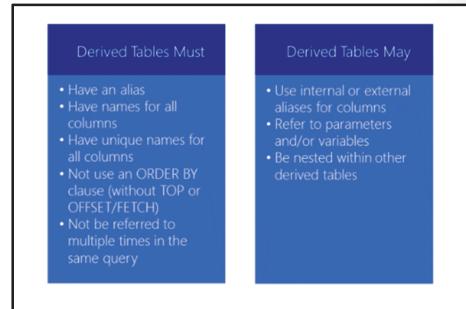
When writing queries that use derived tables, consider the following:

- 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.

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;
```



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 2016 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 might 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.

Nesting and Reusing Derived Tables

Since a derived table is itself a complete query expression, that query can 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 it 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 the CTE discussion later in this module.

Question: How could you rewrite the previous example to eliminate one level of nesting?

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. In Solution Explorer, open the **31 - Demonstration C.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. 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?

Lesson 4

Using CTEs

Another form of table expression provided by SQL Server 2016 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.

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, in addition to 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.

For additional reading on recursive CTEs, see the SQL Server 2016 Technical Documentation:

Recursive Queries Using Common Table Expressions

<http://go.microsoft.com/fwlink/?LinkId=402773>

- 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>;
```

MCT USE ONLY. STUDENT USE PROHIBITED

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. In Solution Explorer, open the **41 - Demonstration D.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Close SQL Server Management Studio without saving any files.

Check Your Knowledge

Question
Which of the following features is required for a CTE query?
Select the correct answer.
<input type="checkbox"/> The query must have a WITH ... AS clause.
<input type="checkbox"/> The query must include a GROUP BY clause.
<input type="checkbox"/> The query must include a CREATE FUNCTION statement.
<input type="checkbox"/> The query must include a nested derived query.

Lab: Using Table Expressions

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server 2016. You have been given 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: **20761B-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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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.
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.

► **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.

► 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

ACT USE ONLY. STUDENT USE PROHIBITED

► **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

► 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.

ACT USE ONLY. STUDENT LICENSE PROHIBITED

► **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.

Module Review and Takeaways

In this module, you have learned how 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.

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

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	12-16
Module Review and Takeaways	12-21

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 use the result of one query to collect the output of a second query, returning the output as a single result set.

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, combining them 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 one set to another; you can then stack result sets from two or more queries into a single output result set. UNION stacks 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 several operators that act on sets, each of which has a different effect on the input sets. The set operators have a number of common features that you need to understand before starting to use them:

- Each input set is the result of a query, which may include any SELECT statement components you have already learned about, except 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 conversion—this may be implicit if the data types support it (using the rules for data type precedence discussed in Module 6 of this course, *Working with SQL Server 2016 Data Types*); otherwise an explicit conversion might be required (using CAST or CONVERT).
- 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; UNION ALL combines all rows, including duplicates. Not all set operators support both forms in SQL Server 2016.

- 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>];
```



Note: When working with set operators, it is 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 should add an ORDER BY to the final results, as you may not use it inside the input queries.

Using the UNION Operator

By using the UNION operator, you can 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. Duplicate rows are eliminated by the UNION operator.

For example, in the TSQL sample database, there are 29 rows in the Production.Suppliers table and 91 rows in the Sales.Customers table.

Combining all rows from each set would yield 29 plus 91—or 120 rows. However, as duplicates that appear in both tables are only returned once, 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)
```

 **Note:** As with all T-SQL statements, remember that no sort order is guaranteed by set operators unless one is explicitly specified. 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 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.

For more information, see *UNION (Transact-SQL)* in the SQL Server 2016 Technical Documentation:

UNION (Transact-SQL)

<http://aka.ms/omv6m7>

- UNION returns a result set of distinct rows combined from both input sets
- Duplicates are removed during query processing (affects performance)



-- only distinct rows from both queries are returned

```
SELECT country, region, city FROM HR.Employees
UNION
SELECT country, region, city FROM Sales.Customers;
```

NOTICE ONLY STUDENT USE PROHIBITED

Using the UNION ALL Operator

The UNION ALL operator works in a similar way to the UNION operator—it combines the two input result sets into one output result set. Unlike UNION, UNION ALL does not filter out duplicate rows.

The following example amends the one from the previous topic, using the UNION ALL operator to combine all supplied locations with all customer locations in the output result set:

- UNION ALL returns a result set with all rows from both input sets
- To avoid the performance penalty caused by filtering duplicates, use UNION ALL over UNION whenever requirements allow it

```
-- 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;
```

Using UNION ALL, 120 rows are returned (29 rows from the Production.Suppliers table and 91 rows from the Sales.Customers table):

country	city
UK	London
USA	New Orleans
...	
Finland	Helsinki
Poland	Warszawa
(120 rows affected)	

As UNION ALL does not perform any filtering of duplicates, UNION ALL should be used in place of UNION in cases where you know there will be no duplicate input rows (or where duplicates exist and are required).

UNION ALL will often run significantly faster than UNION on the same data set; this performance difference becomes more obvious as the number of rows in the input result sets increases.

Demonstration: Using UNION and UNION ALL

In this demonstration, you will see how to:

- Use UNION and UNION ALL.

Demonstration Steps

1. Ensure that the MSL-TMG1, 20761B-MIA-DC, and 20761B-MIA-SQL virtual machines are running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **D:\Demofiles\Mod12\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. At the command prompt, type **y**, and then press Enter.
5. Wait for the script to finish, and then press any key.

6. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
7. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod12\Demo** folder.
8. In Solution Explorer, expand **Queries**, and double-click the **11 - Demonstration A.sql** script file.
9. Select the code under the comment **Step 1**, and then click **Execute**.
10. Select the code under the comment **Step 2**, and then click **Execute**.
11. Select the code under the comment **Step 3**, and then click **Execute**.
12. Keep SQL Server Management Studio open for the next demonstration.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? The results from a UNION query can contain duplicate rows.	

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
When combining the output of two sets, UNION and UNION ALL queries cannot include rows with NULL values, because NULL values cannot be compared.	

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2

Using EXCEPT and INTERSECT

While UNION and UNION ALL combine all rows from input sets, you might 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 might 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 the 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;
```

Returns:

```
country  city
-----  -----
Germany  Berlin
UK       London
Canada   Montréal
France   Paris
Brazil   São Paulo
(5 row(s) affected)
```

- INTERSECT returns the distinct set of rows that appear in both input result sets



```
-- only rows that exist in both queries will be returned
SELECT country, region, city FROM HR.Employees
INTERSECT
SELECT country, region, city FROM Sales.Customers;
```

For more information, see *EXCEPT and INTERSECT (Transact-SQL)* in the SQL Server 2016 Technical Documentation:

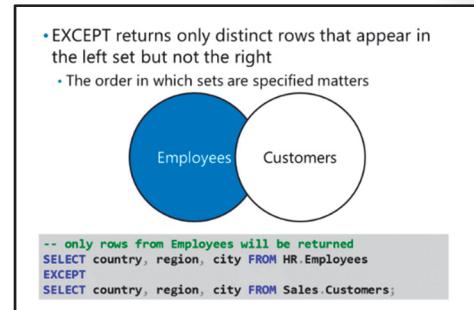
 [EXCEPT and INTERSECT \(Transact-SQL\)](http://aka.ms/u04qu9)

<http://aka.ms/u04qu9>

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 or RIGHT 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 amends the one used previously in this lesson to use EXCEPT to return geographical information that is not common between the Customers table and the Suppliers table. Remember that there are 91 rows in the Customers table and 29 rows 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;
```

There are 24 rows returned. Part of the result set is 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 result sets is reversed:

EXCEPT Example - Input Set Order Reversed

```
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 result sets carefully.

Demonstration: Using EXCEPT and INTERSECT

In this demonstration, you will see how to:

- Use INTERSECT and EXCEPT.

Demonstration Steps

1. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Question
You have a table of employees and a table of customers, both of which contain a column holding the name of the country where the customer or employee is located. You want to know which countries have at least one customer and at least one employee. Which set operator should you use?
Select the correct answer.
UNION ALL
UNION
EXCEPT
INTERSECT
None of the above

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 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.

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; you can pass elements from the left row as input parameters to the TVF.

 **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 in which they appear, relative to one another in the FROM clause.

To use APPLY, you will supply two input sets within a single FROM clause. With APPLY, unlike the set operators you have learned about, the second, or right, table source is logically processed once per row found in the first, or left, table source.

APPLY supports two different forms: CROSS APPLY and OUTER APPLY, which you will learn about in this lesson.

The general syntax for APPLY—each result from the left table source will be passed as an input to the right table source:

APPLY Syntax

```
SELECT <column_list>
FROM <left_table_source> AS <alias>
[CROSS] | [OUTER] APPLY
    <right_table_source> AS <alias>;
```

- APPLY is a table operator used in the FROM clause
- Two forms—CROSS APPLY and OUTER APPLY
- Operates on two input tables, referred to as left and right
- Right table may be any table expression including a derived table or a table-valued function

```
SELECT <column_list>
FROM <left_table_source> AS <alias>
[CROSS] | [OUTER] APPLY
    <right_table_source> AS <alias>;
```

MCT USE ONLY. STUDENT USE PROHIBITED

See Using APPLY in the "Remarks" section of *FROM (Transact-SQL)* in the SQL Server 2016 Technical Documentation:

 **FROM (Transact-SQL)**

<http://aka.ms/r0uc2i>

The CROSS APPLY Operator

As you learned in the previous topic, APPLY executes the right table source for each of the rows in the left table source—and returns the results as a single result set.

The CROSS APPLY form of the operator will include in the output result set only those values from the left table source where a value is found in the right table source.

- CROSS APPLY applies the right table source to each row in the left table source
- Only rows with results in both the left table source and right table source are returned
- Most INNER JOIN statements can be rewritten as CROSS APPLY statements

```
SELECT o.orderid, o.orderdate,
od.productid, od.unitprice, od.qty
FROM Sales.Orders AS o
CROSS APPLY (SELECT productid, unitprice, qty
FROM Sales.OrderDetails AS so
WHERE so.orderid = o.orderid
) AS od;
```

 **Note:** Note that the term CROSS, when used in CROSS APPLY, does not have the same meaning as CROSS when used in CROSS JOIN. Whereas a CROSS JOIN returns all the possible combinations of the left and right table sources, CROSS APPLY returns only the values from the left table source where a value is found in the right table source.

This makes a CROSS APPLY statement very similar to an INNER JOIN—this similarity is such that almost all T-SQL statements that include an INNER JOIN between two tables can be rewritten as a statement using CROSS APPLY.

Consider the following simple SELECT statement, using an INNER JOIN between the sales orders and sales order details tables:

CROSS APPLY; INNER JOIN Example

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

A partial result from the TSQL sample database:

orderid	orderdate	productid	unitprice	qty
10248	2006-07-04 00:00:00.000	11	14.00	12
10248	2006-07-04 00:00:00.000	42	9.80	10
10248	2006-07-04 00:00:00.000	72	34.80	5
10249	2006-07-05 00:00:00.000	14	18.60	9
10249	2006-07-05 00:00:00.000	51	42.40	40
10250	2006-07-08 00:00:00.000	41	7.70	10
10250	2006-07-08 00:00:00.000	51	42.40	35
10250	2006-07-08 00:00:00.000	65	16.80	15
...				
(2155 row(s) affected)				

Here is the same statement rewritten to use CROSS APPLY:

CROSS APPLY; INNER JOIN Rewritten Example

```
SELECT o.orderid, o.orderdate,
od.productid, od.unitprice, od.qty
FROM Sales.Orders AS o
CROSS APPLY ( SELECT productid, unitprice, qty
              FROM Sales.OrderDetails AS so
              WHERE so.orderid = o.orderid
            ) AS od;
```



Note: Notice that the JOIN predicate `Sales.OrderDetails.orderid = Sales.Orders.orderid` moves from the INNER JOIN clause to the WHERE clause of the right table source when the query is rewritten to use CROSS APPLY.

When executed, this query returns the same result as the version written using INNER JOIN:

orderid	orderdate	productid	unitprice	qty
10248	2006-07-04 00:00:00.000	11	14.00	12
10248	2006-07-04 00:00:00.000	42	9.80	10
10248	2006-07-04 00:00:00.000	72	34.80	5
10249	2006-07-05 00:00:00.000	14	18.60	9
10249	2006-07-05 00:00:00.000	51	42.40	40
10250	2006-07-08 00:00:00.000	41	7.70	10
10250	2006-07-08 00:00:00.000	51	42.40	35
10250	2006-07-08 00:00:00.000	65	16.80	15
...				
(2155 row(s) affected)				

The OUTER APPLY Operator

As you learned in an earlier topic, APPLY executes the right table source for each of the rows in the left table source, and returns the results as a single result set.

The OUTER APPLY form of the operator will include all the values from the left table source in the output result set and values from the right table source where they exist. Where the right table source does not contain a value for a left table source value, columns derived from the right table source will have a NULL value.

This makes an OUTER APPLY statement very similar to a LEFT OUTER JOIN—this similarity is such that almost all T-SQL statements that include a LEFT OUTER JOIN between two tables can be rewritten as a statement using OUTER APPLY.

As with LEFT OUTER JOIN, the order in which the table sources appear might influence the result.

- OUTER APPLY applies the right table source to each row in the left table source
- All rows from the left table source are returned—values from the right table source are returned where they exist, otherwise NULL is returned
- Most LEFT OUTER JOIN statements can be rewritten as OUTER APPLY statements

```
SELECT DISTINCT s.country AS supplier_country,
c.country AS customer_country
FROM Production.Suppliers AS s
OUTER APPLY (SELECT country
FROM Sales.Customers AS cu
WHERE cu.country = s.country
) AS c
ORDER BY supplier_country;
```

The following SELECT statement uses a LEFT OUTER JOIN between the suppliers table and the customers table to show all countries where suppliers are located—and which of those countries also contain customers:

OUTER APPLY; LEFT OUTER JOIN Example

```
SELECT DISTINCT s.country AS supplier_country, c.country as customer_country
FROM Production.Suppliers AS s
LEFT OUTER JOIN Sales.Customers AS c
ON c.country = s.country
ORDER BY supplier_country;
```

 **Note:** Notice that the JOIN predicate

```
Sales.Customers.Country = Production.Suppliers.Country
```

moves from the LEFT OUTER JOIN clause to the WHERE clause of the right table source when the query is rewritten to use OUTER APPLY.

This query returns the same result as the LEFT OUTER JOIN version of the query:

supplier_country	customer_country
Australia	NULL
Brazil	Brazil
Canada	Canada
Denmark	Denmark
Finland	Finland
France	France
Germany	Germany
Italy	Italy
Japan	NULL
Netherlands	NULL
Norway	Norway
Singapore	NULL
Spain	Spain
Sweden	Sweden
UK	UK
USA	USA

(16 row(s) affected)

This query can be rewritten using OUTER APPLY:

OUTER APPLY; LEFT OUTER JOIN Rewritten Example

```
SELECT DISTINCT s.country AS supplier_country, c.country as customer_country
FROM Production.Suppliers AS s
OUTER APPLY ( SELECT country
                  FROM Sales.Customers AS cu
                 WHERE cu.country = s.country
            ) AS c
ORDER BY supplier_country;
```

Returns:

```
supplier_country  customer_country
-----
Australia        NULL
Brazil           Brazil
Canada           Canada
Denmark          Denmark
Finland          Finland
France           France
Germany          Germany
Italy             Italy
Japan             NULL
Netherlands      NULL
Norway           Norway
Singapore         NULL
Spain             Spain
Sweden           Sweden
UK                UK
USA               USA
(16 row(s) affected)
```

CROSS APPLY and OUTER APPLY Features

As you learned in the previous topics, there are many similarities between CROSS APPLY and INNER JOIN, and OUTER APPLY and LEFT OUTER JOIN.

However, the APPLY operators enable some types of query to be executed which could not be written using JOINs. These queries rely on the left table source being processed before being applied to the right table source. Two examples shown in this topic are using a query returning top results for each input value and a TVF as the right table source.

A sales manager has requested a report showing the three most recent orders for each customer, including customers with no orders. The following query is one way to meet this requirement:

OUTER APPLY: Three Most Recent Orders Per Customer Example

```
SELECT C.custid, TopOrders.orderid, TopOrders.orderdate
FROM Sales.Customers AS C
OUTER APPLY
    (SELECT TOP (3) orderid, CAST(orderdate AS date) AS orderdate
     FROM Sales.Orders AS O
     WHERE O.custid = C.custid
     ORDER BY orderdate DESC, orderid DESC) AS TopOrders;
```

 **Note:** Note that because OUTER APPLY is used here, customers with no orders are included in the result (with NULL in the orderid and orderdate columns). If CROSS APPLY were used instead of OUTER APPLY, customers with no orders would not appear in the results.

- CROSS APPLY and OUTER APPLY allow query expressions that could not appear in a JOIN to return as part of a single result set
- For example, table-valued functions (TVFs)

```
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;
```

Partial results, including rows with NULLs, appear as follows:

custid	orderid	orderdate
1	11011	2008-04-09
1	10952	2008-03-16
1	10835	2008-01-15
2	10926	2008-03-04
2	10759	2007-11-28
2	10625	2007-08-08
22	NULL	NULL
57	NULL	NULL
58	11073	2008-05-05
58	10995	2008-04-02
58	10502	2007-04-10
(265 row(s) affected)		

A TVF might be used as the right table source for an instance of the APPLY operator.

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: Calling a Table-Valued Function 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;
```



Note: Note that because CROSS APPLY is used here, suppliers with no products are excluded from the result.

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

Demonstration: Using CROSS APPLY and OUTER APPLY

In this demonstration, you will see how to:

- Use forms of the APPLY Operator.

Demonstration Steps

1. In Solution Explorer, open the **31 - Demonstration C.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Test with CROSS APPLY**, and then click **Execute**.
6. Select the code under the comment **Step 4**, and then click **Execute**.
7. Select the code under the comment **Step 5**, and then click **Execute**.
8. Select the code under the comment **Use OUTER APPLY to include customers with no orders**, and then click **Execute**.
9. Close SQL Server Management Studio, without saving any changes.

Question: What is the difference between CROSS APPLY and CROSS JOIN?

Lab: Using Set Operators

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server 2016. 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: **20761B-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 use the UNION operator to write different SELECT statements, and then merge them together into one result.

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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.

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. Staff want to find out the top three products, based on sales revenue, for each customer. Use the APPLY operator to achieve this result.

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 TVF Created for This Lab

► 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**:

```
DROP FUNCTION IF EXISTS 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 more rows are returned 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 TVF Created for This Lab**

1. Remove the created inline TVF by executing the provided T-SQL code:

```
DROP FUNCTION IF EXISTS 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

ACT 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?
5. Close SQL Server Management Studio, without saving any changes.

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

In this module, you have learned about set operators and the APPLY operator.

Review Question(s)

Question: Which set operator would you use to combine sets if you knew there were no duplicates and wanted the best possible performance?

Question: Which form of the APPLY operator will not return rows from the left table if the result of the right table expression is empty?

Question: Which form of the APPLY operator can be used to rewrite LEFT OUTER JOIN queries?

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. After you have learned how to work with windows and window functions, you might find that some types of queries that 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;
```

- Windows extend T-SQL's set-based approach
- Windows allow you to specify an order as part of a calculation, without regard to order of input or final output order
- Windows allow partitioning and framing of rows to support functions
- Window functions can simplify queries that need to find running totals, moving averages, or gaps in data

```
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
Dairy Products	4374	2007	6460
Dairy Products	2689	2008	9149

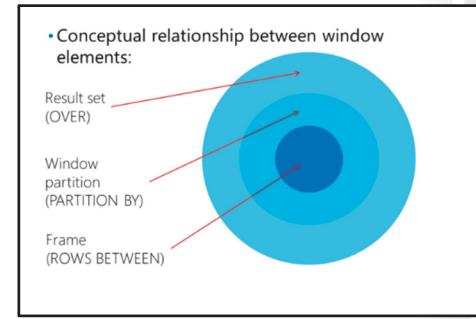
During the next few topics of this lesson, you will learn how to use these query elements.

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:

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.

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

- 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> ]
    )
```

MCT USE ONLY. STUDENT USE PROHIBITED

The next topics will build on this basic use of OVER to define a window of rows.

For further reading on the OVER clause, see the SQL Server 2016 Technical Documentation:

 **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	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

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.

Ordering and Framing

As you have learned, you use window partitions 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.

- Window framing allows you to set start and end boundaries within a window partition
- UNBOUNDED means go all the way to boundary in direction specified by PRECEDING or FOLLOWING (start or end)
- CURRENT ROW indicates start or end at current row in partition
- ROWS BETWEEN allows you to define a range of rows between two points
- Window ordering provides a context to the frame
- Sorting by an attribute enables meaningful position of a boundary
- Without ordering, "start at first row" is not useful because a set has no order

 **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
Dairy Products	4374	2007	6460
Dairy Products	2689	2008	9149

MCT USE ONLY. STUDENT USE PROHIBITED

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 20761D-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **D:\Demofiles\Mod13\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. At the command prompt, type **y**, and then press Enter.
5. Wait for the script to finish, and then press any key.
6. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
7. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod13\Demo** folder.
8. In Solution Explorer, open the **11 - Demonstration A.sql** script file.
9. Select the code under the comment **Step 1**, and then click **Execute**.
10. Select the code under the comment **Step 2**, and then click **Execute**.
11. Select the code under the comment **Step 3**, and then click **Execute**.
12. Select the code under the comment **Step 4**, and then click **Execute**.
13. Select the code under the comment **Step 5**, and then click **Execute**.
14. Keep SQL Server Management Studio open for the next demonstration.

Sequencing Activity

Put the following elements into the logical order in which they are 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

Lesson 2

Exploring Window Functions

SQL Server 2016 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, in addition to others, to operate on a set of rows defined in a window by an OVER clause and its elements.

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;
```

- 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;
```

The partial result:

productid	productname	unitprice	pricerank
-----------	-------------	-----------	-----------

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.

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.

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;
```

- 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;
```

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.

 **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.

- 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.

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 SWNJV	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 the SQL Server 2016 Technical Documentation:

 **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.

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 2016 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, see the SQL Server 2016 Technical Documentation:

 **Analytic Functions (Transact-SQL)**

<http://go.microsoft.com/fwlink/?LinkId=402780>

- 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

Window Offset Functions

Windows offset functions give 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

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. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. Select the code under the comment **Step 6**, and then click **Execute**.
8. Select the code under the comment **Step 7**, and then click **Execute**.
9. Select the code under the comment **Step 8**, and then click **Execute**.
10. Select the code under the comment **Step 9**, and then click **Execute**.
11. Select the code under the comment **Step 10**, and then click **Execute**.
12. Select the code under the comment **Step 11**, and then click **Execute**.
13. Close SQL Server Management Studio without saving any files.

Categorize Activity

Place each windowing function into the appropriate category. Indicate your answer by writing the category number to the right of each item.

Items	
1	SUM()
2	RANK()
3	PERCENT_RANK()
4	MIN()
5	DENSERANK()
6	CUME_DIST()
7	MAX()
8	ROW_NUMBER()
9	PERCENTILE_CONT()
10	NTITLE()
11	PERCENTILE_DISC()

Category 1		Category 2	Category 3
Window Aggregate Functions		Window Ranking Function	Window Distribution Functions

MCIT USE ONLY. STUDENT USE PROHIBITED

Lab: Using Window Ranking, Offset, and Aggregate Functions

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server 2016. 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: **20761B-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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab13\Starter** folder as Administrator.

► **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 in addition to 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:
 - o orderyear as a year of the orderdate column.
 - o 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.

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 achieved 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:
 - o **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.
 - o **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.

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 final task in exercise 2. Write a SELECT statement to retrieve the monthno and val columns. Add two calculated columns:
 - o **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.
 - o **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.
3. Close SQL Server Management Studio without saving any changes.

Results: After this exercise, you should have a basic understanding of how to use window aggregate functions in T-SQL statements.

ACT USE ONLY
STUDENT USE PROHIBITED

Module Review and Takeaways

In this module, you have learned how 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.

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?

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	14-12
Module Review and Takeaways	14-18

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 projecting 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)

- Pivoting data is rotating data from a rows-based orientation to a columns-based orientation
- Distinct values from a single column are projected across as headings for other columns—may include aggregation

Category	2006	2007	2008
Dairy Products	12	2006	
Grains/Cereals	10	2006	
Dairy Products	5	2006	
Produce	9	2006	
Produce	40	2006	
Seafood	10	2006	
Produce	35	2006	
Condiments	15	2006	
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	
			1842 3998 3694
			962 2895 1441
			10 12 12
			2089 4374 2669
			2626 1377
			950 2189 1060
			549 1583 858
			1286 3679 2716

Pivoted data

To analyze this by category and year, you might 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.

For more information, see the SQL Server 2016 Technical Documentation:

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:

- **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.

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)

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, that are not used for aggregation or spreading, will be used as grouping elements—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.

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



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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **D:\Demofiles\Mod14\Setup.cmd** as an administrator.
3. At the command prompt, type **y**, and then press Enter.
4. When the script completes, close the command prompt window.
5. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
6. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod14\Demo** folder.
7. In Solution Explorer, open the **11 - Demonstration A.sql** script file.
8. Select the code under the comment **Step 1**, and then click **Execute**.
9. Select the code under the comment **Step 2**, and then click **Execute**.
10. Select the code under the comment **Step 3**, and then click **Execute**.
11. Select the code under the comment **Step 4**, and then click **Execute**.
12. Select the code under the comment **Step 5**, and then click **Execute**.
13. Select the code under the comment **Step 6**, and then click **Execute**.
14. Select the code under the comment **Step 7**, and then click **Execute**.
15. 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?

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 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
);
```

- 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
);
```

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, in addition to the empty parentheses notation to aggregate all rows:

GROUPING SETS Example

```
Code Example Content
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 because neither Category nor Cust take part in the aggregation.

For more information, see Using GROUP BY with ROLLUP, CUBE, and GROUPING SETS in the SQL Server 2016 Technical Documentation:

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.

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 ().

- 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;
```

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.

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;
```

- 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;
```

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 might 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, see the SQL Server 2016 Technical Documentation:

 [GROUPING_ID \(Transact-SQL\)](#)

<http://go.microsoft.com/fwlink/?LinkId=402787>

SQL Server also provides a GROUPING function, which accepts only one input to return a bit. For more information, see *GROUPING (Transact-SQL)* in the SQL Server 2016 Technical Documentation:

 [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. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. Select the code under the comment **Step 6**, and then click **Execute**.
8. Select the code under the comment **Step 7**, and then click **Execute**.
9. Select the code under the comment **Step 8**, and then click **Execute**.
10. 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.

Lab: Pivoting and Grouping Sets

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server 2016. You have been given 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: **20761B-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 required 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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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 given you 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.
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:
 - o 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.
 - o You will need to JOIN tables Sales.Orders, Sales.OrderDetails, Production.Products, and Production.Categories.
 - o 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.
 - o 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

1. 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).
2. 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
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:
 - o Calculated column with the alias groupid (use the GROUPING_ID function with the order year and order month as the input parameters).
 - o Year of the orderdate column as orderyear.
 - o Month of the orderdate column as ordermonth.
 - o Total sales value using the val column as salesvalue.
 - o 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.
3. Close SQL Server Management Studio without saving any changes.

Results: After this exercise, you should have an understanding of how to use the GROUPING SETS, CUBE, and ROLLUP subclauses in T-SQL statements.

Module Review and Takeaways

In this module, you have learned how to:

- Write queries that pivot and unpivot result sets.
- Write queries that specify multiple groupings with grouping sets.

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?

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	15-15
Module Review and Takeaways	15-21

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 result 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.

Because 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 might be useful to note that stored procedures can also include INSERT, UPDATE, DELETE, and other valid T-SQL commands. They can also 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 might only use the provided application to access data rather than other tools.

Stored procedures also offer other benefits, including network and database engine performance improvements. See the course 20762B: *Developing Microsoft SQL Server Databases* for additional information on these benefits and more details on creating and using stored procedures.

For more information, see the SQL Server 2016 Technical Documentation:



<http://aka.ms/sshz88>

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:

- 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.

- 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;
```



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.

For more information, see the SQL Server 2016 Technical Documentation:



Execute a Stored Procedure

<http://aka.ms/Dyplvh>

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **D:\Demofiles\Mod15\Setup.cmd** as an administrator.
3. At the command prompt, type **y**, and press Enter.
4. Wait until the script completes, and then press Enter.
5. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
6. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod15\Demo** folder.
7. Open the **11 - Demonstration A.sql** script file.
8. Select the code under the comment **Step 1**, and then click **Execute**.
9. Select the code under the comment **Step 2**, and then click **Execute**.
10. Select the code under the comment **Step 3**, and then click **Execute**.
11. Select the code under the comment **Step 4**, and then click **Execute**.
12. Select the code under the comment **Step 5**, and then click **Execute**.
13. Select the code under the comment **Step 6**, and then click **Execute**.
14. 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?

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 might 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.

To call a stored procedure and pass parameters, use the following syntax:

- 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;
```

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 that 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.

For more information about passing parameters to stored procedures, see the SQL Server 2016 Technical Documentation:

Specify Parameters

<http://aka.ms/R5zjio>

Working with OUTPUT Parameters

So far in this module, you have seen procedures that return results through an embedded SELECT statement. SQL Server also gives you 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;
```

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. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. In Object Explorer, expand **Databases**, expand **AdventureWorks**, expand **Programmability**, and then expand **Stored Procedures**.
4. Expand any procedure, expand **Parameters**, and then point out list of parameters, data type and direction.
5. Select the code under the comment **Step 3**, and then click **Execute**.
6. Select the code under the comment **Step 4**, and then click **Execute**.
7. Select the code under the comment **Step 5**, and then click **Execute**.
8. Select the code under the comment **Step 6**, and then click **Execute**.
9. Select the code under the comment **Step 7**, and then click **Execute**.
10. Select the code under the comment **Step 8**, and then click **Execute**.
11. Select the code under the comment **Step 9**, and then click **Execute**.
12. Select the code under the comment **Step 10**, and then click **Execute**.
13. Select the code under the comment **Step 11**, and then click **Execute**.
14. 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 the @empid parameter.

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, in addition to 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 might include any of the data manipulations you have already learned in this course.

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

- Stored procedures can be wrappers for simple or complex SELECT statements
- Procedures may include input and output parameters in addition to 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

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, see the SQL Server 2016 Technical Documentation:

Modify a Stored Procedure

<http://aka.ms/Bn33te>

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 ...
```

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. In Solution Explorer, open the **31 - Demonstration C.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. 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.

MCT ICE ONLY. STUDENT USE PROHIBITED

Lesson 4

Working with Dynamic SQL

In organizations where creating parameterized stored procedures is not supported, you might 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 might 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, in addition to input parameters.

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.

For more information, see the *Using EXECUTE with a Character String in the EXECUTE (Transact-SQL)* topic in the SQL Server 2016 Technical Documentation:

EXECUTE (Transact-SQL)

<http://aka.ms/Fors3z>

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
```

To use sp_executesql with parameters, provide the query code, in addition to 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.

• Using sp_executesql

- Accepts string as code to be run
- Supports input, output parameters for query
- Allows parameterized code while minimizing risk of SQL injection
- Can perform better than EXEC due to query plan reuse

```
DECLARE @sqlcode AS NVARCHAR(256) =
      N'<code_to_run>';
EXEC sys.sp_executesql @statement = @sqlcode;
```

```
DECLARE @sqlcode AS NVARCHAR(256) =
      N'SELECT GETDATE() AS dt';
EXEC sys.sp_executesql @statement = @sqlcode;
```

ACT USE ONLY STUDENT USE PROHIBITED

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, see the SQL Server 2016 Technical Documentation:

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. In Solution Explorer, open the **41 - Demonstration D.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. 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?

Lab: Executing Stored Procedures

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server 2016. You have been given 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: **20761B-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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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`:

```
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;
```

3. Write a T-SQL statement to execute the created procedure.
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\52 - Lab Exercise 1 - Task 1 Result.txt**.
5. What is the difference between the previous T-SQL code and this one?
6. 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 required 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;
```

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?

► **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

► 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

- 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

In this module, you have learned how 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.

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

Module 16

Programming with T-SQL

Contents:

Module Overview	16-1
Lesson 1: T-SQL Programming Elements	16-2
Lesson 2: Controlling Program Flow	16-8
Lab: Programming with T-SQL	16-12
Module Review and Takeaways	16-18

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 might 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, because 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—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. There is a complete list in the SQL Server 2016 Technical Documentation, where you will also find additional reading.



Batches

<http://go.microsoft.com/fwlink/?LinkId=402796>

- T-SQL batches are collections of one or more T-SQL statements sent to SQL Server as a unit for parsing, optimization, and execution
- Batches are terminated with GO by default
- Batches are boundaries for variable scope
- Some statements (for example, CREATE FUNCTION, CREATE PROCEDURE, CREATE VIEW) may not be combined with others in the same batch

```
CREATE VIEW <view_name>
AS ...;
GO
CREATE PROCEDURE <procedure_name>
AS ...;
GO
```

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.

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 because the entire batch would be rejected.

- 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

MERGE 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
```



Variables (Transact-SQL)

<http://aka.ms/Jub7kl>

Working with Variables

Once you have declared a variable, you must initialize it, or assign it a value. You can 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.

- Variables are objects that allow storage of a value for use later in the same batch
- Variables are defined with the DECLARE keyword
 - In SQL Server 2008 and later, variables can be declared and initialized in the same statement
- Variables are always local to the batch in which they're declared and go out of scope when the batch ends

```
--Declare and initialize variables  
DECLARE @numrows INT = 3, @catid INT = 2;  
--Use variables to pass parameters to procedure  
EXEC Production.ProdsByCategory  
    @numrows = @numrows, @catid = @catid;  
GO
```

- Initialize a variable using the DECLARE statement

```
DECLARE @i INT = 0;
```
- Assign a single (scalar) value using the SET statement

```
SET @i = 1;
```
- Assign a value to a variable using a SELECT statement
 - Be sure that the SELECT statement returns exactly one row

```
SELECT @i = COUNT(*) FROM Sales.SalesOrderHeader;
```

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 might have synonyms defined for them include tables, views, stored procedures, and user-defined functions.

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.

- 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;
```

 **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.

For more information, see *Using Synonyms (Database Engine)* in the SQL Server 2016 Technical Documentation:

 **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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **D:\Demofiles\Mod16\Setup.cmd** as an administrator.
3. At the command prompt, type **y**, and then press Enter.
4. When the script completes, close the command prompt window.
5. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
6. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod16\Demo** folder.
7. Open the **11 - Demonstration A.sql** script file.
8. Select the code under the comment **Step 1**, and then click **Execute**.
9. Select the code under the comment **Step 2**, and then click **Execute**.
10. Select the code under the comment -- **Create a proc to search for category**, and then click **Execute**.
11. Select the code under the comment -- **Set up table for batch demos**, and then click **Execute**.
12. Select the code under the comment **Step 3**, and then click **Execute**.
13. Select the code under the comment -- **Show that the batch was successful**, and then click **Execute**.
14. Select the code under the comment **Step 4**, and then click **Execute**.
15. Select the code under the comment **Step 5**, and then click **Execute**. Note the error message.
16. Select the code under the comment --**Show that no rows were inserted**, and then click **Execute**.
17. Select the code under the comment **Step 6**, and then click **Execute**.
18. Select the code under the comment --**Run the following batch in its entirety to show the choices**, and then click **Execute**.
19. Select the code under the comment **Step 7**, and then click **Execute**.
20. Select the code under the comment -- **Declare a parameter to search for category**, and then click **Execute**.
21. Select the code under the comment -- **Test it locally**, and then click **Execute**.
22. Select the code under the comment **Step 8**, and then click **Execute**.
23. Select the code under the comment **Step 9**, and then click **Execute**.
24. Select the code under the comment **Step 10**, and then click **Execute**.
25. Select the code under the comment **Step 11**, and then click **Execute**.
26. 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?

Lesson 2

Controlling Program Flow

All programming languages include elements that help 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 multistatement user-defined functions. These control-of-flow elements mean you can programmatically determine whether or not to execute statements and programmatically determine the order of those statements that should be executed.

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 providing 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.

For more information, see the SQL Server 2016 Technical Documentation:

 **Control-of-Flow Language (Transact-SQL)**

<http://aka.ms/Pvhnn>

- SQL Server provides additional language elements that control the flow of execution of T-SQL statements
 - Used in batches, stored procedures, and multistatement 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
```

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.

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;
```

For more information, see the SQL Server 2016 Technical Documentation:

IF...ELSE (Transact-SQL)

<http://aka.ms/mvl2f5>

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
```

NOTICE 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.

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;
```

- 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.
```

 **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, see the SQL Server 2016 Technical Documentation:

 **WHILE (Transact-SQL)**

<http://aka.ms/Beqqci>

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. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. 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 need 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	

Lab: Programming with T-SQL

Scenario

As a junior database developer for Adventure Works, you have so far focused on writing reports using corporate databases stored in SQL Server 2016. 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: **20761B-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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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.ssmssqln** 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 3 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

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:
 - o For @i variable values less than 5, set the value of the @result variable to "Less than 5".
 - o For @i variable values between 5 and 10, set the value of the @result variable to "Between 5 and 10".
 - o For all @i variable values over 10, set the value of the @result variable to "More than 10".
 - o 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

- 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.

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**

- 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.

Module Review and Takeaways

In this module, you have learned how 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.

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 yet exist?

Question: Will a WHILE loop exit when the predicate evaluates to NULL?

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	17-15

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 works in T-SQL. 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.

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:

- **Error number.** Unique number identifying the specific error.
- **Error message.** Text describing the error.
- **Severity.** Numeric indication of seriousness from 1 to 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.

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 to 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 messages are in `sys.messages`
• Add custom application errors using `sp_add_message`

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 predefined, 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 inline where they are generated, or you can predefine them in the `sys.messages` table alongside the system-supplied errors. The error numbers used for custom errors must be 50001 or above.

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';
```

In addition to 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 because 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

The ability to raise errors in T-SQL makes error handling in the application easier, because 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.

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

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.messages.
- Errors raised by THROW only cause transaction abort when used in conjunction with SET XACT_ABORT ON and the session is terminated.

- SQL Server provides the THROW statement
- Successor to the RAISERROR statement
- Does not require defining errors in the sys.messages table

```
THROW 50001, 'An Error Occurred', 0;
```

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 returns last error code
- Can be captured and stored in a variable

@@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.

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 might want 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

- Ensure that the 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
- Run **D:\Demofiles\Mod17\Setup.cmd** as an administrator.
- At the command prompt, type **y**, and then press Enter.
- When the script completes, close the command prompt window.
- Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
- Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod17\Demo** folder.
- Open the **11 - Demonstration A.sql** script file.
- Select the code under the comment **Step 1**, and then click **Execute**.
- Select the code under the comment **Step 2**, and then click **Execute**.
- Select the code under the comment **--Capture @@ERROR into a variable**, and then click **Execute**.
- Select the code under the comment **--Create a custom error message**, and then click **Execute**.
- Select the code under the comment **--Use a custom error message**, and then click **Execute**.
- 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?

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 noncatchable 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 means you can 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 are often used to release resources implicitly. There is no equivalent FINALLY block in T-SQL.

- TRY block defined by BEGIN TRY...END TRY statements
 - Place all code that might raise an error between them
 - No code may be placed between END TRY and BEGIN CATCH
 - TRY and CATCH blocks may be nested
- CATCH block defined by BEGIN CATCH...END CATCH
 - Execution moves to the CATCH block when catchable errors occur within the TRY block

MCT USE ONLY. STUDENT USE PROHIBITED

Error Handling Functions

CATCH blocks make the error-related information available throughout the duration of the CATCH block. This includes subscopes, such as stored procedures, run from within the CATCH block.

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.

- CATCH blocks make the error-related information available throughout the duration of the CATCH block
- @@ERROR is reset when the next statement is run

Catchable vs. Noncatchable 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.

Catchable vs. Noncatchable 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.

- TRY/CATCH blocks will only catch errors in the same block
- Common examples of noncatchable errors are:
 - Compile errors, such as syntax errors, that prevent a batch from compiling
 - Statement level recompilation issues that usually relate to deferred name resolution

Common Noncatchable Errors

Common examples of noncatchable errors are:

- Compile errors, such as syntax errors, that prevent a batch from compiling.
- Statement level recompilation issues that 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.

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 throwing the original error to the client application, so that it can be handled there.

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.

- 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;
```

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 might 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. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Close SQL Server Management Studio without saving any files.

NOTICE ONLY. STUDENT USE PROHIBITED

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?

Lab: Implementing Error Handling

Scenario

As a junior database developer for Adventure Works, you will be creating stored procedures using corporate databases stored in SQL Server 2012. 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: **20761B-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa\$\$w0rd**

Exercise 1: Redirecting Errors with TRY/CATCH

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. 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.

► **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 given you 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');
```

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.

Results: After this exercise, you should be able to capture and handle errors using a TRY/CATCH construct.

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. Rethrow 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: Rethrow 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 given you 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**

- 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

In this module, you have learned how to:

- Implement T-SQL error handling.
- Implement structured exception handling.

Review Question(s)

Question: Which error types cannot be 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

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 should 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 T-SQL commands that begin, commit, or roll back work, based on user-issued code. T-SQL 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
```

- A transaction is a group of tasks defining a unit of work
- The entire unit must succeed or fail together—no partial completion is permitted

--Two tasks that make up a unit of work

```
INSERT INTO Sales.Orders ...
INSERT INTO Sales.OrderDetails ...
```

- Individual data modification statements are automatically treated as stand-alone transactions
- User transactions can be managed with T-SQL commands:
 - BEGIN/ COMMIT/ ROLLBACK TRANSACTION
- SQL Server uses locking mechanisms and the transaction log to support transactions

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 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 might include locks and entries in the transaction log to allow SQL Server to undo changes made by the transaction, should a rollback be required.

For more information, see the SQL Server 2016 Technical Documentation:

 **Transaction Statements (Transact-SQL)**

<http://aka.ms/H9jd4y>

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.

The following example illustrates this problem:

- 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

```
--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;
```

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 after 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.

ACT USE ONLY STUDENT LICENSE 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.

The following example shows the addition of T-SQL 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.

- 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;
```

Demonstration: Transactions and the Database Engine

In this demonstration, you will see how to:

- Use transactions.

Demonstration Steps

Use Transactions

1. Ensure that the 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
2. Run **D:\Demofiles\Mod18\Setup.cmd** as an administrator.
3. At the command prompt, type **y**, and then press Enter.
4. When the script completes, close the command prompt window.
5. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
6. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod18\Demo** folder.
7. Open the **11 - Demonstration A.sql** script file.
8. Select the code under the comment **Step 1**, and then click **Execute**.
9. Select the code under the comment **Step 2**, and then click **Execute**.
10. Select the code under the comment **Step 3**, and then click **Execute**. Note the error message.
11. Select the code under the comment **Step 4**, and then click **Execute**.
12. Select the code under the comment **Step 5**, and then click **Execute**.
13. Keep SQL Server Management Studio open for the next demonstration.

Categorize Activity

Place 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 TRANSACTION
2	BEGIN TRY
3	COMMIT TRANSACTION
4	END TRY
5	END TRANSACTION
6	BEGIN CATCH
7	ROLLBACK TRANSACTION
8	END CATCH
9	INSERT
10	RAISERROR

Category 1		Category 2
TCL		Non-TCL

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2

Controlling Transactions

To control how SQL Server treats your data modification statements, you need to use T-SQL 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. For more information on locking in SQL Server, see course 20762B: *Developing Microsoft SQL Server 2016 Databases*.

For more information on BEGIN TRANSACTION statements, see the SQL Server 2016 Technical Documentation:

 **BEGIN TRANSACTION (Transact-SQL)**

<http://aka.ms/E3u6jb>

For more information on nested transactions, see the SQL Server 2016 Technical Documentation:

 **Nesting Transactions**

<http://go.microsoft.com/fwlink/?LinkId=402857>

COMMIT TRANSACTION

When 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
```

- COMMIT ensures all of the transaction's modifications are made a permanent part of the database
- COMMIT frees resources, such as locks, used by the transaction
- In your T-SQL code, if a transaction is successful, commit it

```
BEGIN TRY
    BEGIN TRAN -- marks beginning of work
    INSERT INTO Sales Orders ...
    INSERT INTO Sales OrderDetails ...
    COMMIT TRAN -- mark the work as complete
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.

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, in addition to objects created. ROLLBACK also allows SQL Server to release resources, such as locks, held during the transaction's lifespan.

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 might 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.

- 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.

```

NOTICE ONLY STUDENT USE PROHIBITED

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;
```

For more information, see *Transaction Statements (Transact-SQL)* in the SQL Server 2016 Technical Documentation:

 **Transaction Statements (Transact-SQL)**

<http://aka.ms/h9jd4y>

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.

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.

For more information, see the SQL Server 2016 Technical Documentation:

 **SET XACT_ABORT (Transact-SQL)**

<http://aka.ms/Qehdh1>

- 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;
```

Demonstration: Controlling Transactions

In this demonstration, you will see how to:

- Control transactions.

Demonstration Steps

Control Transactions

1. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Select the code under the comment **Step 5**, and then click **Execute**.
7. Select the code under the comment **Step 6**, and then click **Execute**. Note the error message.
8. Select the code under the comment **Step 7**, and then click **Execute**.
9. Select the code under the comment **Step 8**, and then click **Execute**.
10. 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 writer is now receiving errors resulting from locks on database records. What can you do to troubleshoot this problem?

Lab: Implementing Transactions

Scenario

As a junior database developer for Adventure Works, you will be creating stored procedures using corporate databases stored in SQL Server 2016. 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: **20761B-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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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.

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. In this case, the default behavior would be two transactions because 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);
```

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 be 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.

NOTICE ONLY STUDENT USE PROHIBITED

6. After the written SELECT statement, write a ROLLBACK statement to cancel the transaction. Only execute the ROLLBACK statement.
7. Highlight this and execute the written SELECT statement against the HR.Employees table again.
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**

- 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.

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**

- 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**

- 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

In this module, you have learned how 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.

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?

Course Evaluation

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.

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 2016

Lab: Working with SQL Server 2016 Tools

Exercise 1: Working with SQL Server Management Studio

► Task 1: Open Microsoft SQL Server Management Studio

1. Ensure that the **20761B-MIA-DC** and **20761B-MIA-SQL** virtual machines are running.
2. Log on to **20761B-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa\$\$w0rd**.
3. Start SQL Server Management Studio.
4. In the **Connect to Server** dialog box, click **Cancel**.
5. Close the Object Explorer window by clicking the **Close** icon.
6. Close the Solution Explorer window by clicking the **Close** icon.
7. To open the Object Explorer pane, on the **View** menu, click **Object Explorer** (or press F8).
8. To open the Solution Explorer pane, on the **View** menu, click **Solution Explorer** (or press Ctrl+Alt+L).

► Task 2: Configure the Editor Settings

1. In SQL Server Management Studio, on the **Tools** menu, click **Options**.
2. In the **Options** dialog box, expand the **Environment** option, and then click **Fonts and Colors**.
3. In the **Show settings for** list, click **Text Editor**.
4. In the **Size** box, set the font size to **14**.
5. In the left pane, expand **Text Editor**, expand **Transact-SQL**, and then click **IntelliSense**.
6. In the **Transact-SQL IntelliSense Settings** section, clear the **Enable IntelliSense** check box.
7. In the left pane, under **Transact-SQL**, click **Tabs**, and then change the **Tab size** to **6**.
8. In the left pane, expand **Query Results**, expand **SQL Server**, and then click **Results to Grid**.
9. Select the **Include column headers when copying or saving the results** check box, and then click **OK**.

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, point to **New**, and then click **Project**.
2. In the **New Project** dialog box, click **SQL Server Scripts**.
3. In the **Name** box, type **MyFirstProject**.
4. In the **Location** box, type **D:\Labfiles\Lab01\Starter**, and then click **OK** to create the new project.
5. In Solution Explorer, under **MyFirstProject**, right-click the **Queries** folder, and then click **New Query**.

6. In the **Connect to Database Engine** dialog box, click **Cancel**.
7. Under the **Queries** folder, right-click **SQLQuery1.sql**, click **Rename**, type **MyFirstQueryFile**, and then press Enter.
8. On the **File** menu, click **Save All**.

► **Task 2: Add an Additional Query File**

1. In Solution Explorer, right-click the **Queries** folder, and then click **New Query**.
2. In the **Connect to Database Engine** dialog box, click **Cancel**.
3. In the **Queries** folder, right-click **SQLQuery1.sql**, click **Rename**, type **MySecondQueryFile**, and then press Enter.
4. On the taskbar, click **File Explorer**.
5. In File Explorer, navigate to the **D:\Labfiles\Lab01\Starter\MyFirstProject\MyFirstProject** folder to see where the files have been created.
6. In SQL Server Management Studio, in Solution Explorer, right-click **MySecondQueryFile.sql**, and then click **Remove**.
7. In the **Microsoft SQL Server Management Studio** dialog box, click **Remove**.
8. In File Explorer, press F5 to refresh, notice that the file **MySecondQueryFile.sql** is still there.
9. In SQL Server Management Studio Solution, in Solution Explorer, right-click **MyFirstQueryFile.sql**, and then click **Remove**.
10. In the **Microsoft SQL Server Management Studio** dialog box, click **Delete**.
11. In File Explorer, press F5 to refresh, notice that the **MyFirstQueryFile.sql** file has been deleted.

► **Task 3: Reopen the Created Project**

1. In SQL Server Management Studio Solution, on the **File** menu, click **Save All**.
2. On the **File** menu, click **Exit** to close the project and SQL Server Management Studio.
3. Open SQL Server Management Studio.
4. In the **Connect to Server** dialog box, click **Cancel**.
5. On the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab01\Starter\MyFirstProject** folder, click **MyFirstProject.ssmssln**, and then click **Open**.
7. In File Explorer, navigate to the **D:\Labfiles\Lab01\Starter\MyFirstProject\MyFirstProject** folder.
8. Drag the **MySecondQueryFile.sql** file to the **Queries** folder in **Solution Explorer**.
9. On the **File** menu, click **Save All**.

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

Exercise 3: Using SQL Server 2016 Technical Documentation

► Task 1: Launch SQL Server 2016 Technical Documentation

1. On the Start page, type **manage help settings**, and then click **Manage Help Settings**.
2. In the **Help Library Manager - Microsoft Help Viewer 1.0** dialog box, click **Choose online or local help**.
3. On the **Settings** page, under **Set your preferred help experience**, click **I want to use online help**, and then click **OK**.
4. If **I want to use online help** is already selected, press **Cancel**.
5. On the **Settings** page, click **Exit** to close the Help Library Manager.

► Task 2: Use SQL Server 2016 Technical Documentation

1. On the Start page, type **sql server documentation**, and then click **SQL Server Documentation**.
2. If the **Online Help Consent** dialog box appears, click **Yes**.
3. Browse the help article, and then close Internet Explorer.

Results: After this exercise, you will understand how to find the information you need in SQL Server 2016 Technical Documentation.

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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.
4. Press any key to close the command window.

► Task 2: Execute the T-SQL Script

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, ensure Windows Authentication is selected, and then click **Connect**.
3. On the **File** menu, point to **Open**, and then click **Project/Solution**.
4. In the **Open Project** dialog box, browse to the **D:\Labfiles\Lab02\Starter\Project** folder, and then double-click **Project.ssmssqln**.
5. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
6. When the query window opens, click **Execute**. 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 text under the **Task 2** description:

```
SELECT firstname
      ,lastname
      ,city
      ,country
  FROM HR.Employees;
```

 **Note:** To highlight it, 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**. It is very important to understand that you can highlight a specific part of the code inside the T-SQL script, and execute only that part. 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.

MCT USE ONLY. STUDENT USE PROHIBITED

3. On the **File** menu, click **Close**.
4. Close SQL Server Management Studio, without saving any changes.

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

► Task 1: Execute the T-SQL Script

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, and then click **Options**.
3. On the **Connection Properties** tab, in the **Connect to database** list, ensure **<default>** is selected, and then click **Connect**.
4. On the **File** menu, point to **Open**, and then click **Project/Solution**.
5. In the **Open Project** dialog box, browse to the **D:\Labfiles\Lab02\Starter\Project** folder, and then double-click **Project.ssmssln**.
6. In Solution Explorer, expand **Queries**, and then double-click **61 - Lab Exercise 2.sql**.
7. In the query pane, click **Execute**.
8. Notice that you get the error message:

```
Msg 208, Level 16, State 1, Line 18
Invalid object name 'HR.Employees'.
```

Why do you think this happened? This error is very common when you are beginning to learn T-SQL.

9. 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: Change the Database Context with the GUI

1. In the **Available Databases** list, click **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 country column. Only rows for which the logical expression evaluates to **TRUE** are returned by the **WHERE** phase to the subsequent logical query processing phase.

MCT USE ONLY. STUDENT USE PROHIBITED

► **Task 3: Change the Database Context with T-SQL**

1. In the script **61 - Lab Exercise 2.sql**, find the lines:

```
--USE TSQl;
--Go
```

2. Delete the first two characters, so that the line looks like this:

```
USE TSQl;
GO
```

3. By deleting these two characters, you have removed the comment mark. Now the line will not be ignored by SQL Server.
4. On the **File** menu, click **Save 61 - Lab Exercise 2.sql**.
5. On the **File** menu, click **Close**. This will close the T-SQL script.
6. In Solution Explorer, double-click **61 - Lab Exercise 2.sql**.
7. Click **Execute**.
8. 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 sets the database context to the **TSQl** database. The next statement in the T-SQL script, the **SELECT**, then executes against the **TSQl** database.
9. On the **File** menu, click **Close**.

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 Initial 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 them.

► **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;
```

6. Click **Execute**. The database context is now set to the **TSQl** database.

7. Highlight the statement:

```
SELECT
firstname, lastname, city, country
FROM HR.Employees
WHERE country = 'USA'
ORDER BY lastname;
```

8. Click **Execute**.

9. 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. You will also have an appreciation of how to order the results of a query.

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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**.
4. When the script has finished, press Enter.

► Task 2: View All the Tables in the ADVENTUREWORKS Database in Object Explorer

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, and then click **Options**.
3. Under **Connection Properties**, in the **Connect to database** list, click **<Browse server>**.
4. In the **Browse for Databases** dialog box, click **Yes**.
5. In the **Browse Server for Databases** dialog box, under **User Databases**, click **TSQL**, and then click **OK**.
6. In the **Connect to Server** dialog box, on the **Login** tab, in the **Authentication** list, click **Windows Authentication**, and then click **Connect**.
7. In Object Explorer, under **MIA-SQL**, expand **Databases**, expand **TSQL**, and then expand **Tables**.
8. 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 That Returns All Rows and Columns from a Table

1. On the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab03\Starter\Project** folder, and then double-click **Project.ssmssln**.
3. In Solution Explorer, expand **Queries**, and then double-click **Lab Exercise 1.sql**.
4. In the query window, highlight the statement **USE TSQL;** and then click **Execute**.
5. In the query pane, after the **Task 2** description, type the following query:

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

6. Highlight the query you typed, and click **Execute**.
7. In the query pane, type the following code after the first query:

```
SELECT *  
FROM
```

8. In Object Explorer, under **MIA-SQL**, under **Databases**, under **TSQL**, under **Tables**, click **Sales.Customers**.
9. Drag the selected table into the query pane, after the FROM clause. Add a semicolon to the end of the SELECT statement. Your finished query should look like this:

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

10. Highlight the written query, and click **Execute**.

► **Task 4: Write a SELECT Statement That Returns Specific Columns**

1. In Object Explorer, expand **Sales.Customers**, expand **Columns** and observe all the columns in the **Sales.Customers** table.
2. In the query pane, after the **Task 3** description, type the following query:

```
SELECT contactname, address, postalcode, city, country  
FROM Sales.Customers;
```

3. Highlight the written query, and click **Execute**.
4. 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 **Lab Exercise 2.sql**.
2. In the query window, highlight the statement **USE TSQL;** and then click **Execute**.

3. In the query pane, after the **Task 1** description, type the following query:

```
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.

► **Task 2: Write a SELECT Statement That Uses the DISTINCT Clause**

1. Highlight the previous query, and then on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 2** description.
3. On the **Edit** menu, click **Paste**. You have now copied the previous query to the same query window after the task 2 description.
4. Modify the query by typing **DISTINCT** after the **SELECT** clause. Your query should look like this:

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

5. Highlight the written query, and click **Execute**.
 6. 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;
```

*If all combinations of values in the city and region columns in the **Sales.Customers** table are unique, both queries would return the same number of rows. 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 just apply to the first column in the list.*

Results: After this exercise, you should understand 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 **Lab Exercise 3.sql**.
2. In the query window, highlight the statement **USE TSQl;**, and click **Execute**.
3. In the query pane, after the **Task 1** description, type the following query:

```
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, after the **Task 2** description, type the following query:

```
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 with 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 Table and Column Aliases

1. In the query pane, after the **Task 3** description, type the following query:

```
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 best practice to always use AS when specifying aliases—then 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 **Lab Exercise 4.sql**.
2. In the query window, highlight the statement **USE TSQL**; and then click **Execute**.
3. In the query pane, after the **Task 1** description, type the following query:

```
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, after the **Task 2** description, type the following:

```
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, and then on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 3** description.
3. On the **Edit** menu, click **Paste**. You have now copied the previous query to the same query window after the task 3 description.

4. 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;
```

5. Highlight the written query, and click **Execute**.
6. 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.

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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**.
4. At the command prompt, type **y**, press Enter, wait for the script to finish, and then press any key.

► Task 2: Write a SELECT Statement That Uses an Inner Join

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, and then click **Connect**.
3. On the **File** menu, point to **Open**, and then click **Project/Solution**.
4. In the **Open Project** dialog box, browse to the **D:\Labfiles\Lab04\Starter\Project** folder, and then double-click **Project.ssmssln**.
5. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
6. In the query window, highlight the statement **USE TSQL;**, and click **Execute**.
7. In the query pane, after the **Task 1** description, type the following query:

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

8. Highlight the written query, and click **Execute**.
9. 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 this is the predicate because the 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.

Exercise 2: Writing Queries That Use Multiple-Table Inner Joins

► Task 1: Execute the T-SQL Statement

- In Solution Explorer, double-click the **61 - Lab Exercise 2.sql** query.
- In the query window, highlight the statement **USE TSQL;**, and then click **Execute**.
- Under the **Task 1** description, highlight the written query, and click **Execute**.
- Observe the error message:
Ambiguous column name 'custid'.
- This error occurred because the custid column appears in both tables; 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

- Highlight the previous query, and on the **Edit** menu, click **Copy**.
- In the query window, click the line after the **Task 2** description, and on the **Edit** menu, click **Paste**.
- 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;
```

- Highlight the modified query and click **Execute**.

► Task 3: Change the Table Aliases

- Highlight the previous query, and on the **Edit** menu, click **Copy**.
- In the query window, click the line after the **Task 3** description, and on the **Edit** menu, click **Paste**.
- 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;
```

- Highlight the written query and click **Execute**.
- Compare these 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 the error messages:

```
Msg 4104, Level 16, State 1, Line 57
The multipart identifier "Customers.custid" could not be bound.
Msg 4104, Level 16, State 1, Line 57
The multipart identifier "Customer.contactname" could not be bound.
Msg 4104, Level 16, State 1, Line 57
The multipart 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;
```

10. Highlight the written query and click **Execute**.

► Task 4: Add an Additional Table and Columns

1. In the query pane, after the **Task 4** description, type the following query:

```
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.

Exercise 3: Writing Queries That Use Self Joins

► Task 1: Write a Basic SELECT Statement

1. In Solution Explorer, double-click the **71 - Lab Exercise 3.sql** query.
2. In the query window, highlight the statement **USE TSQL;**, and then click **Execute**.
3. In the query pane, after the **Task 1** description, type the following query:

```
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, and on the **Edit** menu, click **Copy**.
2. In the query window, after the **Task 2** description, click the line, and on the **Edit** menu, click **Paste**.
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?
6. 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.

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 **81 - Lab Exercise 4.sql** query.
2. In the query window, highlight the statement **USE TSQL;** and then click **Execute.**
3. In the query pane, after the **Task 1** description, type the following query:

```
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 **91 - Lab Exercise 5.sql** query.
2. In the query window, highlight the statement **USE TSQL;** and then click **Execute.**
3. Under the **Task 1** description, highlight the T-SQL code 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, after the **Task 2** description, type the following query:

```
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 3,294 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 366 (3,294/9) rows in the HR.Calendar table.

► **Task 3: Drop the HR.Calendar Table**

- Under the **Task 3** description, highlight the written query 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: Write Queries that Filter Data Using a WHERE Clause

► Task 1: Prepare the Lab Environment

1. Ensure that the 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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.
4. At the command prompt, when prompted, press any key.

► Task 2: Write a SELECT Statement Using 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, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab05\Starter\Project** folder, and then double-click **Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
5. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
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. Highlight the query and click **Execute**.

Note the use of the “N” prefix for the character literal ‘Brazil’. 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. If the “N” is omitted, then the query may still run successfully. However, the safest way is to include the “N” every time, to ensure the results are predictable. You will learn more about data types in the next module.

► Task 3: Write a SELECT Statement Using 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 query and click **Execute**.

► **Task 4: Write a SELECT Statement Using 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 4a** description, and click **Execute**.
2. Highlight the provided T-SQL statement, and on the toolbar, click **Edit**, and then click **Copy**.
3. In the query window, click the line after the **Task 4b** description, and on the toolbar, click **Edit**, and then click **Paste**.

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 where 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 nonpreserved 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 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. Highlight the written query and click **Execute**.

It is important to note that, when you are looking for a NULL, you should use the IS NULL operator, 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.

The choice of which attribute to filter from the nonpreserved side of the join is also important. You should choose an attribute that can have a NULL only 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.

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: Write Queries that Sort Data Using an ORDER BY Clause

► Task 1: Write a SELECT Statement Using an ORDER BY Clause

1. In Solution Explorer, double-click **61 - Lab Exercise 2.sql**.
2. In the query pane, 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;
```

4. Highlight the written query and click **Execute**.

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. However, 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. Data type precedence and working with date values are covered in detail in the next module.

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.

► **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'.
```

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, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 3a** description, and on the **Edit** menu, click **Paste**.
3. Modify the T-SQL statement to remove the WHERE clause, and add 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. Highlight the previous query, and on the **Edit** menu, click **Copy**.
6. In the query window, click the line after the **Task 3b** description, and on the **Edit** menu, click **Paste**.
7. 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;
```

8. Highlight the written query and click **Execute**.
9. Compare the results for Task 3a and 3b.
10. Why were you equally able to use a source column name or an alias column name?

Results: After this exercise, you should know how to use an ORDER BY clause.

Exercise 3: Write 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 **71 - Lab Exercise 3.sql**.
2. In the query pane, highlight the statement **USE TSQL;**, and then 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 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. Highlight the query and click **Execute**.

Remember that the OFFSET-FETCH clause was a new functionality in SQL Server 2012 and will not work in earlier versions. Unlike the TOP clause, the OFFSET-FETCH clause must be used with the ORDER BY clause.

► 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. Highlight the query and click **Execute**.

Implementing this task with the OFFSET-FETCH clause is possible but not easy because, unlike TOP, OFFSET-FETCH does not support a PERCENT option.

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.

Exercise 4: Write 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 **81 - Lab Exercise 4.sql**.
2. In the query pane, highlight the statement **USE TSQL;** and then 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 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 query and click **Execute**.

► Task 3: Write a Generic Form of the OFFSET-FETCH Clause for Paging

1. The correct code is:

```
OFFSET (@pagenum - 1) * @pagesize ROWS FETCH NEXT @pagesize ROWS ONLY
```

2. To test the above expression, type the following query after the **Task 3** description:

```
DECLARE @pagenum int,
        @pagesize int;
SET @pagenum = 3;
SET @pagesize = 10;
SELECT
    custid, orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate, orderid
OFFSET (@pagenum - 1) * @pagesize ROWS FETCH NEXT @pagesize ROWS ONLY;
```

3. Highlight the query and click **Execute**.
4. Compare your results with the recommended results in file **D:\Labfiles\Lab05\Solution\84 - Lab Exercise 4- Task 3 Result**. Try changing the values for **@pagenum** and/or **@pagesize**, highlight the whole query (including the **DECLARE** and **SET** statements) and then click **Execute**.

Results: After this exercise, you will be able to use OFFSET-FETCH to work page-by-page through a result set returned by a SELECT statement.

Module 6: Working with SQL Server 2016 Data Types

Lab: Working with SQL Server 2016 Data Types

Exercise 1: Writing Queries That Return Date and Time Data

► Task 1: Prepare the Lab Environment

1. Ensure that the 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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**.
4. Wait for the script to finish, and when prompted, press any key.

► Task 2: Write a SELECT Statement to Retrieve Information About the Current Date

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab06\Starter\Project** folder, and then double-click **Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
5. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
6. In the query pane, after the **Task 1** description, type the following query:

```
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**.

MCIT USE ONLY. STUDENT LICENCE PROHIBITED

► **Task 3: Write a SELECT Statement to Return the Date Data Type**

1. In the query pane, after the **Task 2** description, type the following queries:

```
SELECT DATEFROMPARTS(2015, 12, 11) AS somedate;
SELECT CAST('20151211' AS DATE) AS somedate;
SELECT CONVERT(DATE, '12/11/2015', 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, after the **Task 3** description, type the following query:

```
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, 1, EOMONTH(CURRENT_TIMESTAMP, -1)) AS firstday;
```

2. Highlight the written query, and click **Execute**.

► **Task 5: Write a SELECT Statement to Show Whether a Table of Strings Can Be Used as Dates**

1. Under the **Task 4** description, highlight the written query, 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 cast to the specified data type if the casting succeeds; otherwise, it returns NULL. Don't 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 formats '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 Customers with Orders in a Given Month**

1. In Solution Explorer, double-click **61 - Lab Exercise 2.sql**.
2. In the query pane, highlight the statement **USE TSQL**, and then click **Execute**.
3. In the query pane, after the **Task 1** description, type the following query:

```
SELECT DISTINCT
    custid
    FROM Sales.Orders
    WHERE
    YEAR(orderdate) = 2008
    AND MONTH(orderdate) = 2;
```

4. Highlight the written query, and click **Execute**.

Note that, as a performance enhancement, you could also write a query that uses a range format that would utilize an index on Sales.Orders.orderdate. 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, after the **Task 2** description, type the following query:

```
SELECT
    CURRENT_TIMESTAMP AS currentdate,
    DATEADD (day, 1, EOMONTH(CURRENT_TIMESTAMP, -1)) AS firstofmonth,
    EOMONTH(CURRENT_TIMESTAMP) AS endofmonth;
```

2. Highlight the written query, and click **Execute**.

This query uses the EOMONTH function, which was added in SQL Server 2012.

► **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, after the **Task 3** description, type the following query:

```
SELECT
    orderid, custid, orderdate
    FROM Sales.Orders
    WHERE
    DATEDIFF(
        day,
        orderdate,
        EOMONTH(orderdate)
    ) < 5;
```

2. Highlight the written query, and click **Execute**.

► **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, after the **Task 4** description, type the following query:

```
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 **71 - Lab Exercise 3.sql**.
2. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
3. In the query pane, after the **Task 1** description, type the following query:

```
SELECT
CONCAT(contactname, N' (city: ', city, N')') AS contactwithcity
FROM Sales.Customers;
```

4. Highlight the written query, and click **Execute**.

An alternate way to write this query would be to use the + (plus) operator:

```
SELECT
contactname + N' (city: ' + city + N')' AS contactwithcity
FROM Sales.Customers;
```

► **Task 2: Add an Additional Column to the Concatenated String Which Might Contain NULL**

1. In the query pane, after the **Task 2** description, type the following query:

```
SELECT
CONCAT(contactname, N' (city: ', city, N', region: ', region, N')') AS fullcontact
FROM Sales.Customers;
```

2. Highlight the written query, and click **Execute**.

An alternative way to write this query would be to use the + (plus) operator, which requires the COALESCE function to replace a NULL with an empty string. Later modules will include more examples of how to handle NULL.

```
SELECT
contactname + N' (city: ' + city + N', region: ' + COALESCE(region, '') + N')' AS
fullcontact
FROM Sales.Customers;
```

► **Task 3: Write a SELECT Statement to Retrieve Customer Contacts Based on the First Character in the Contact Name**

1. In the query pane, after the **Task 3** description, type the following query:

```
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 **81 - Lab Exercise 4.sql**.
2. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
3. In the query pane, after the **Task 1** description, type the following query:

```
SELECT
contactname,
SUBSTRING(contactname, 0, CHARINDEX(N',', contactname)) AS lastname
FROM Sales.Customers;
```

4. Highlight the written query, and click **Execute**.

► **Task 2: Write a Query to Retrieve the Contact's First Name Using SUBSTRING**

1. In the query pane, after the **Task 2** description, type the following query:

```
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 Format the Customer ID**

1. In the query pane, after the **Task 3** description, type the following query:

```
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**.

An alternative way to write this query would be to use the FORMAT function. The query would then look like this:

```
SELECT custid,
FORMAT(custid, N'\C00000') AS custnewid
FROM Sales.Customers;
```

► **Task 4: Challenge: Write a SELECT Statement to Return the Number of Character Occurrences**

1. In the query pane, after the **Task 4** description, type the following query:

```
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**.
3. Close SQL Server Management Studio without saving any files.

Results: After this exercise, you should have an understanding of how to use the character functions.

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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**.
4. When the script has finished, press Enter.

► 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, point to **Open**, and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab07\Starter\Project\Project.ssmssqln**.
4. In Solution Explorer, expand **Queries**, and double-click **01 Setup.sql**.
5. On the toolbar, click **Execute**. When the script has executed, you should get the messages indicating (9 row(s) affected), (88 row(s) affected) and (3 row(s) affected).
6. Close the 01 Setup.sql pane and open a new query window by clicking the **New Query** icon.
7. When the query window opens, type **USE TempDB**, followed by **GO** on the next line, and then, on the toolbar, click **Execute**.

8. Below the GO statement in the open window, type the following query:

```
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'
);
```

9. Click **Execute**.

10. Make sure the row has been inserted, and then close the window. You will be asked if you want to save the query—you can choose where to save it and what to call it.

► Task 3: Insert a Row with a SELECT Statement As the Data Provider

1. Click **New Query**.
2. In the query pane, type the following query:

```
USE TempDB
GO
INSERT INTO Sales.Customers
(
    Companyname
    , contactname
    , contacttitle
    , address
    , city
    , region
    , postalcode
    , country
    , phone
    , fax
)
SELECT
    Companyname
    , contactname
    , contacttitle
    , address
    , city
    , region
    , postalcode
    , country
    , phone
    , fax
    FROM dbo.PotentialCustomers;
```

3. Click **Execute**.
4. Make sure that the rows have been inserted, and then close the window. You will be asked if you want to save the query—you can choose where to save it and what to call it.

How could you have checked the data as it was transferred by the query? Remember the OUTPUT command? If not, you can look at the exercise solution labeled **42 - Lab Exercise 1b Solution.sql**.

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:

```
Use TempDB
GO
UPDATE Sales.Customers
SET contacttitle='Sales Consultant'
WHERE city='Berlin'AND contacttitle='Sales Representative';
```

3. Click **Execute**.
4. Make sure the rows have been modified, and then close the window. You will be asked if you want to save the query—you can choose where to save it and what to call it.

How could you have checked the data as it was transferred by the query? Remember the OUTPUT command? If not, you can look at the exercise solution labeled **43 - Lab Exercise 2 Solution.sql**.

► Task 2: Delete Rows

1. Click **New Query**.
2. In the query pane, type the following query:

```
USE TempDB
GO
DELETE FROM dbo.PotentialCustomers
WHERE contactname
IN( 'Taylor, Maurice','Mallit, Ken', 'Tiano, Mike' );
```

3. Click **Execute**.
4. Make sure the rows have been deleted, and then close the window. You will be asked if you want to save the query—you can choose where to save it and what to call it.

How could you have checked the data as it was transferred by the query? Remember the OUTPUT command? If not, you can look at the exercise solution labeled **44 - Lab Exercise 2b Solution.sql**.

Results: After successfully completing this exercise, you will have updated all the records in the Customers table that have a city of Berlin and a contacttitle of Sales Representative, to now 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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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**.
4. At the command prompt, type **y**, and then press Enter.
5. Wait for the script to finish, and press Enter.

► **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, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab08\Starter\Project** folder, and then double-click **Project.ssmssqln**.
4. In Solution Explorer, expand the **Queries** folder, and then double-click **51 - Lab Exercise 1.sql**.
5. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
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**.

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**.

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.

3. 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;
```

4. Highlight the written query, and click **Execute**. Observe the result. The rows that could not be converted have a NULL.

Results: After this exercise, you should be able to use conversion functions.

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 **61 - Lab Exercise 2.sql**.
2. In the query pane, highlight the statement **USE TSQL**, and then 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;
```

The IIF function was new in SQL Server 2012. It was added mainly to support migrations from Microsoft Access to SQL Server. You can use the CASE expression to achieve the same result.

4. 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. In the query pane, 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. 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);
```

3. Highlight the modified 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.

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab09\Starter\Project** folder, and then double-click **Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
5. In the query pane, highlight the statement **USE TSQL;**, and click **Execute**.
6. In the query pane, type the following query after the **Task 2** 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, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 3** description, and on the **Edit** menu, click **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, there will be 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 4** 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 5** 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**.



Note: 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.

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 **61 - Lab Exercise 2.sql**.
2. In the query pane, highlight the statement **USE TSQL;**, and then 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, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 2** description, and on the **Edit** menu, click **Paste**.
3. Modify the T-SQL statement so that it adds extra columns. Your query should look like this:

```
SELECT
    o.orderid, o.orderdate,
    SUM(d.qty * d.unitprice) AS salesamount,
    COUNT(*) AS nooforderlines,
    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**.

► **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 **71 - Lab Exercise 3.sql**.
2. In the query pane, highlight the statement **USE TSQL**; and then 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, because 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. To get the correct number of distinct customers, you can 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 **81 - Lab Exercise 4.sql**.
2. In the query pane, highlight the statement **USE TSQL;**, and then 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**.

► **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, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 3** description, and on the **Edit** menu, click **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**.
3. Close SQL Server Management Studio without saving any files.

Results: After this exercise, you should have an understanding of how to use the HAVING clause.

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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.
4. At the command prompt, press any key.

► **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, point to **Open** and click **Project/Solution**.
3. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab10\Starter\Project** folder, and then double-click **Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
5. In the query pane, highlight the statement **USE TSQL**, and click **Execute**.
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**.

► **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 Multiresult Subqueries

► Task 1: Write a SELECT Statement to Retrieve Specific Products

1. In Solution Explorer, double-click **61 - Lab Exercise 2.sql**.
2. In the query pane, highlight the statement **USE TSQL**, and then 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**, and on the **Edit** menu, click **Copy**.
3. In the query window, under the **Task 3** description, click the line after the provided T-SQL statement, and on the **Edit** menu, click **Paste**.
4. Highlight the written query, and click **Execute**.

5. Notice that you have an empty result despite having two rows when you first ran the query in task 2. Why did you have 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 multiresult 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 **71 - Lab Exercise 3.sql**.
2. In the query pane, highlight the statement **USE TSQL;**, and then 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**.

► **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 exist. 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**

- Under the **Task 5** description, highlight the provided T-SQL statement, and then 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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab11\Starter\Project** folder, and then double-click **Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
5. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
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**.

► **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 then click **Execute**.
2. Observe the error message:

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 then click **Execute**.
2. Observe the error message:

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**

- 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 **61 - Lab Exercise 2.sql**.
2. In the query pane, highlight the statement **USE TSQl;**, and then 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**.

► **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 **71 - Lab Exercise 3.sql**.
2. In the query pane, highlight the statement **USE TSQL**, and then 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**.

► **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.

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 **81 - Lab Exercise 4.sql**.
2. In the query pane, highlight the statement **USE TSQL;**, and then 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**

- 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

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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**, wait for the script to finish, and then press any key.

► 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, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab12\Starter\Project** folder, and then double-click **Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
5. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
6. In the query pane, after the **Task 1** description, type the following query:

```
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, after the **Task 2** description, type the following query:

```
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.

► **Task 4: Merge the Results from Task 1 and Task 2**

1. In the query pane, after the **Task 3** description, type the following query:

```
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, and on the **Edit** menu, click **Copy**.
5. In the query window, click the line after the written T-SQL statement, and on the **Edit** menu, click **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, after the **Task 4** description, type the following query:

```

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 **61 - Lab Exercise 2.sql**.
2. In the query pane, highlight the statement **USE TSQl;**, and then click **Execute**.
3. In the query pane, after the **Task 1** description, type the following query:

```

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**.

► **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**. Note that the query retrieves 265 rows.

► **Task 3: Use the OUTER APPLY Operator**

1. Highlight the previous query in **Task 2**, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 3** description, and on the **Edit** menu, click **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. Observe the result to see 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**, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 4** description, and on the **Edit** menu, click **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 TVF Created for This Lab**

- 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**

- In Solution Explorer, double-click **71 - Lab Exercise 3.sql**.
- In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
- In the query pane, after the **Task 1** description, type the following query:

```
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;
```

- 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**

- In the query pane, after the **Task 2** description, type the following query:

```
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;
```

- Highlight the written query, and click **Execute**.

► **Task 3: Write a SELECT Statement to Retrieve Customers Who Spent More Than \$10,000**

- In the query pane, after the **Task 3** description, type the following query:

```
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**.

► **Task 4: Write a SELECT Statement That Uses the EXCEPT and INTERSECT Operators**

1. Highlight the query from **Task 2**, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 4** description, and on the **Edit** menu, click **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 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**, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 5** description, and on the **Edit** menu, click **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 are always processed first.
7. Close SQL Server Management Studio, without saving any changes.

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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**.
5. When the query window opens, highlight the statement **USE TSQL**; and click **Execute**.
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**.
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**.

MCT USE ONLY. STUDENT LICENCE PROHIBITED

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**.
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**.

► **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**.
 3. Close SQL Server Management Studio without saving any changes.

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

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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**.
5. In the query window, highlight the statement **USE TSQL**; and click **Execute**.
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**.

► **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**.

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.

► **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 because 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 that columns 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. In the query window, 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**

- Highlight the provided T-SQL statement after the **Task 3** description and click **Execute**.

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

► **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. In the query window, 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**.

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**.
3. Close SQL Server Management Studio without saving any changes.

Results: After this exercise, you should have an understanding of how to use the GROUPING SETS, CUBE, and ROLLUP subclauses in T-SQL statements.

MCIT USE ONLY. STUDENT USE PROHIBITED

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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 **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
5. In the query window, highlight the statement **USE TSQL;** and click **Execute** on the toolbar.
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.

► **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.

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**.
 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.

► 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.

► **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. In the query window, 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**.

► **Task 2: Execute the Stored Procedure sys.sp_help**

1. In the query pane, type the following T-SQL code after the **Task 2** description:

```
EXEC sys.sp_help 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**

- 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.

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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.ssmssqln**.
4. In Solution Explorer, expand **Queries**, and then double-click the query **51 - Lab Exercise 1.sql**.
5. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
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 + ' ' + lastname FROM HR.Employees WHERE empid = 1);
SELECT @empname AS employee;
```

2. Highlight the written T-SQL code and click **Execute**.
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 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. 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".
```

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.

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. In the query window, 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**.

► **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.

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. In the query window, 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. In the query window, 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.

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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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**.
5. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
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.

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**.

► **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**.

Results: After this exercise, you should be able to capture and handle errors using a TRY/CATCH construct.

Exercise 2: Using THROW to Pass an Error Message Back to a Client

► **Task 1: Rethrow 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**.

► **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**

- 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 20761B-MIA-DC and 20761B-MIA-SQL virtual machines are both running, and then log on to 20761B-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**.
5. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
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);
```

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. In the query window, 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;
```

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;
```

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.