Microsoft

OFFICIAL MICROSOFT LEARNING PRODUCT

20761A

Querying Data with Transact-SQL

Product Number: 20761A

Part Number (if applicable): X20-96724

Released: 04/2016

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.

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

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

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

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

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

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

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

   provided you comply with the following:
   iii. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
   iv. you will ensure each End User attending an Authorized Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Authorized Training Session,
   v. you will ensure that each End User provided with the hard-copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,
   vi. you will ensure that each Trainer teaching an Authorized Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Authorized Training Session,

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

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

Elle s'applique également, même si Microsoft 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 performance[1]. Ask your instructor about Microsoft Certification exam promotions and discounts that may be available to you.

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

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

Sincerely,

Microsoft Learning
www.microsoft.com/learning

Microsoft | Learning

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

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 2-15

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

Lab: Introduction to T-SQL Querying 2-22

Module Review and Takeaways 2-25

Module 3: Writing SELECT Queries

Module Overview 3-1

Lesson 1: Writing Simple SELECT Statements 3-2

Lesson 2: Eliminating Duplicates with DISTINCT 3-6

Lesson 3: Using Column and Table Aliases 3-11

Lesson 4: Writing Simple CASE Expressions 3-16

Lab: Writing Basic SELECT Statements 3-19

Module Review and Takeaways 3-25

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 6-21

Lab: Working with SQL Server 2016 Data Types 6-27

Module Review and Takeaways 6-33

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 7-13

Lab: Using DML to Modify Data 7-16

Module Review and Takeaways 7-19

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 8-14

Lesson 4: Using Functions to Work with NULL 8-18

Lab: Using Built-in Functions 8-22

Module Review and Takeaways 8-26

Module 9: Grouping and Aggregating Data

Module Overview 9-1

Lesson 1: Using Aggregate Functions 9-2

Lesson 2: Using the GROUP BY Clause 9-10

Lesson 3: Filtering Groups with HAVING 9-16

Lab: Grouping and Aggregating Data 9-20

Module Review and Takeaways 9-26

Module 10: Using Subqueries

Module Overview 10-1

Lesson 1: Writing Self-Contained Subqueries 10-2

Lesson 2: Writing Correlated Subqueries 10-7

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

Lab: Using Subqueries 10-14

Module Review and Takeaways 10-19

Module 11: Using Set Operators

Module Overview 11-1

Lesson 1: Writing Queries with the UNION Operator 11-2

Lesson 2: Using EXCEPT and INTERSECT 11-6

Lesson 3: Using APPLY 11-10

Lab: Using Set Operators 11-17

Module Review and Takeaways 11-22

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 Set Operators L11-1

About This Course

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

Course Description

Note: This first release ('A') MOC version of course 20761A has been developed on prerelease software (CTP3.3). Microsoft Learning will release a 'B' version of this course after the RTM version of the software is available.

The main purpose of this 3 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

- 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 Set Operators"

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**: 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, please go to www.microsoft.com/learning/help. To inquire about the Microsoft Certification Program, send an email to mcphelp@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
20761A-MIA-DCA	Domain controller for the ADVENTUREWORKS domain.
20761A-MIA-SQL	SQL Server and SharePoint Server

Software Configuration

The following software is installed:

- Microsoft Windows Server 2012 R2
- Microsoft SQL Server 2016 CTP3.3
- 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 20761A-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*
- 8GB or higher
- 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.

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, services, and how databases are structured. This will help you understand how SQL Server works before you start writing SQL Server queries.

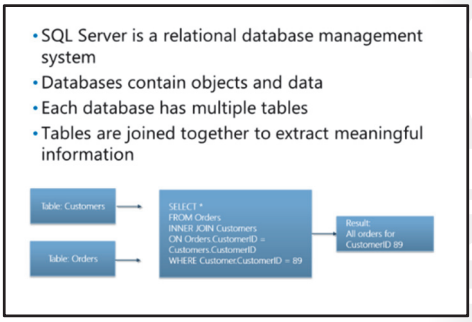
Lesson Objectives

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 we mean 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, and one thing only. 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. SQL Server enables you to get this information by relating these tables to one another. We 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 we 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:

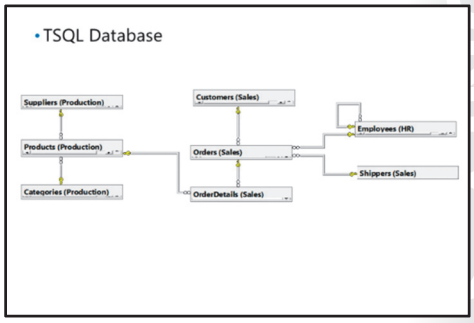
- **master**: the system configuration database.
- **model**: the template database. SQL Server will apply any changes made in model to new databases.
- **msdb**: used by SQL Server Agent to schedule jobs and alerts.
- **tempDb**: a temporary store for data such as work tables. This database is dropped 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:



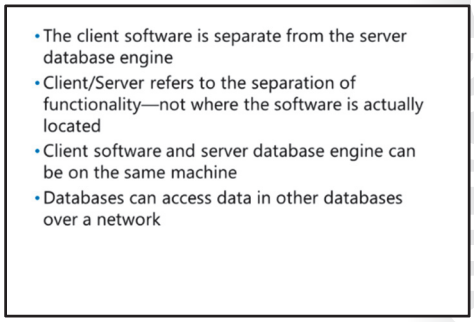
- **Schemas**. These are logical containers for tables and views.
- **Tables**. These mostly relate to one another using Foreign Key constraints.
- **Views**. These display information from more than one table.

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.

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.

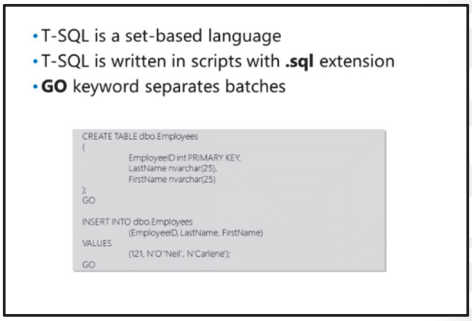
Wherever the client and server software is located, it makes no difference to the way you write T-SQL code. On the logon screen, you just specify the SQL Server you want to connect to.

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.

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®.

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

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.

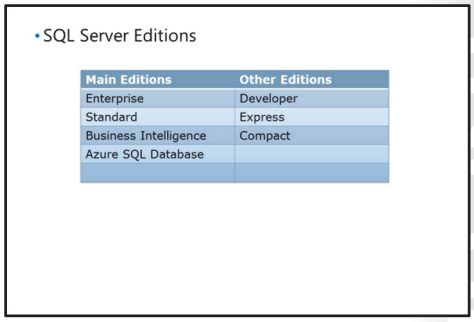
SQL Server 2016 is a major new release with enhanced security, support for hybrid cloud installations, and major improvements in the analytics functionality.

Note: Although the name may suggest otherwise, SQL Server 2008 R2 is not a service pack for SQL Server 2008. SLQ Server 2008 R2 is an independent version (number 10.5) with enhanced multi-server 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 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

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.

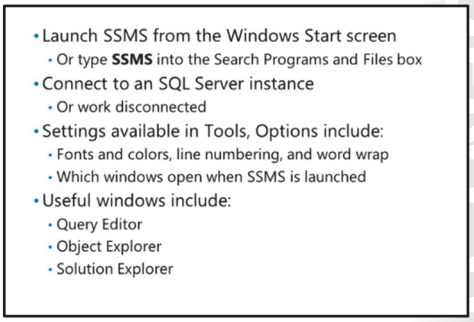
Lesson Objectives

After completing this lesson you will be able to:

- Start SSMS.
- Use SSMS to connect to on-premises SQL Server instances.
- Explore an 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 with 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.

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 an 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.

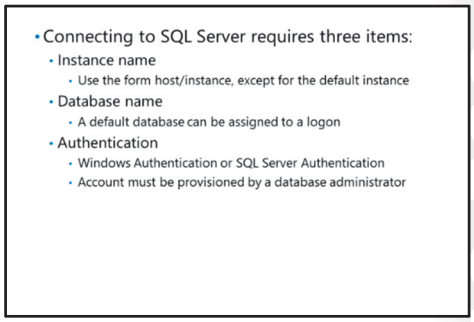
For more information on using SSMS, see "Use SQL Server Management Studio" in SQL Server 2016 Technical Documentation.

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:

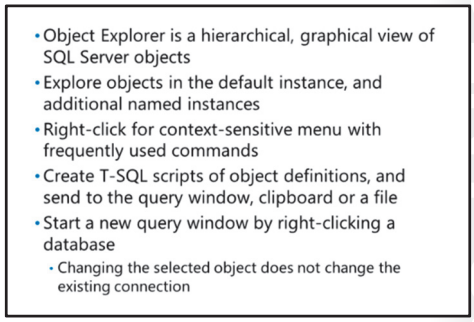


- The instance you want to connect to. This must be in the format: **hostname\instancename**.
- For example, MIA-SQL\Proseware would connect to the Proseware instance on the Windows server named MIA-SQL. If you are connecting to the default instance, you may omit the instance name.
- The name of the 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?

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 on an object such as a table. The available options include query and script generators for object definitions.



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.

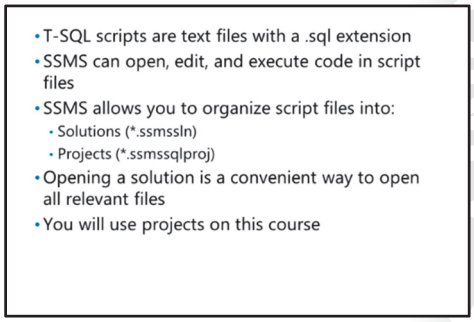
7. To avoid typing, drag an object from the Object Explorer hierarchy into the query window.

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. This enables you to 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 Windows Explorer.



Object	Parent	Description
Solution	-	A solution is a conceptual container for projects. Solutions have a **.ssmssln** 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.

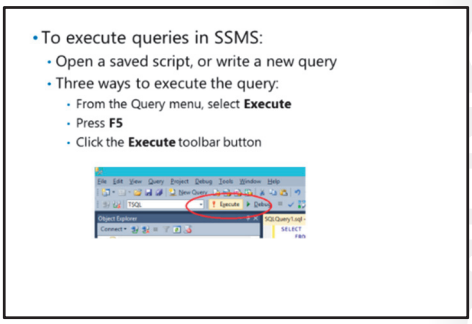
To create a new script that will be stored as part of a project, right-click the **Queries** folder in the **Project** and select **New Query**.

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, hold the Alt key as you drag to move the file.

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 **.ssmssln** 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 **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.

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.

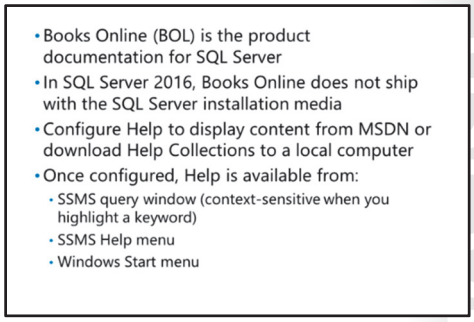
Additional Reading: For a list of keyboard shortcuts available in SSMS, see MSDN "SQL Server Management Studio Keyboard Shortcuts".

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.



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

SQL Server 2016 Technical Documentation

http://aka.ms/dxlgjb

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 **20761A-MIA-DC** and **20761A-MIA-SQL** virtual machines are running.
2. Log on to **20761A-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd.**
3. In the **D:\Demofiles\Mod01\Setup.cmd** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
4. Click **Yes** when you are prompted to confirm that you want to run the command file, 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. If the **Object Explorer** pane is not visible, on the **View** menu, click **Object Explorer**.
2. In **Object Explorer**, expand the **Databases** folder to see a list of databases.
3. Expand the **TSQL** database.
4. Expand the **Tables** folder.
5. Expand the **Sales.Customers** table.
6. Expand the b folders.
7. View the list of columns, and the data type information for each column.
8. Note the data type for the **CompanyName** column.

Work with T-SQL scripts

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

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

10. Select the code and click **Execute** on the toolbar.
11. Point out the results pane.
12. Click **File** and then click **Save All**.
13. Click **File**, and then click **Close Solution**.
14. On the **File** menu, point to **Recent Projects and Solutions**, and then click **Module 1 Demonstration.ssmssln**.
15. Point out the **Solution Explorer** pane.

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 provided you with 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 SQL Server 2016 Technical Documentation.

Estimated Time: 30 minutes

Virtual machine: **20761A-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 Windows 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 Delete.)
4. Check in Windows Explorer 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 Windows 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 Windows Explorer to the Queries folder beneath **MyFirstProject** in SSMS Solution Explorer. (If the Solution Explorer window is not visible, enable it as you did in Exercise 1.)
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

1. 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

Review Question(s)

Question: Can an 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 an SQL Server Management Studio solution contain?

Module 2

Introduction to T-SQL Querying

Contents:

Module Overview 2-1

Lesson 1: Introducing T-SQL 2-2

Lesson 2: Understanding Sets 2-12

Lesson 3: Understanding Predicate Logic 2-15

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

Lab: Introduction to T-SQL Querying 2-22

Module Review and Takeaways 2-25

Module Overview

Transact-SQL, or T-SQL, is the language 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

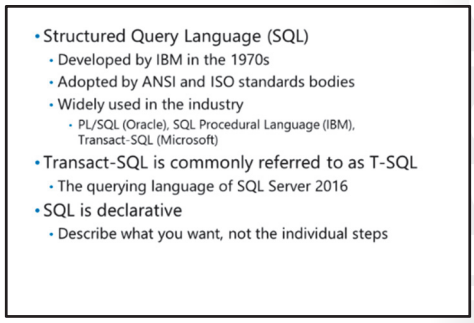
In this lesson, you will learn the role of T-SQL in writing SELECT statements. You will also learn about many of the T-SQL language elements and which ones are useful for writing queries.

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.

The ANSI standard has gone through several revisions, including SQL-89 and SQL-92, whose specifications are either fully or partly supported by T-SQL. SQL Server 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
```

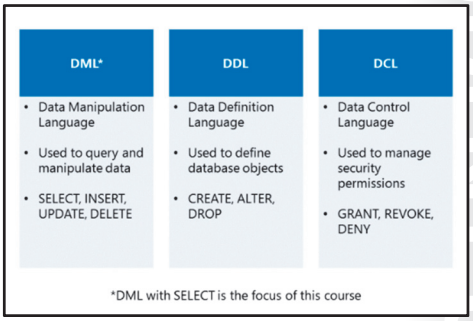
With T-SQL, the SQL Server 2016 database engine will determine the optimal path to access the data and return a matching set. Your role is to learn to write efficient and accurate T-SQL code so you can properly describe the set you wish to retrieve.

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 Microsoft Official Course 20473-2*: Administering a SQL Database Infrastructure*.

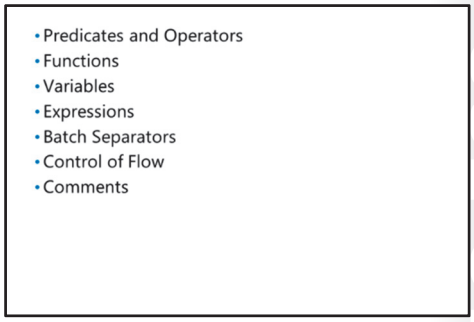
For additional information on DML, DDL, and DCL commands, see Books Online at:

Transact-SQL Reference (Database Engine)

http://aka.ms/hjmhuj

T-SQL Language Elements

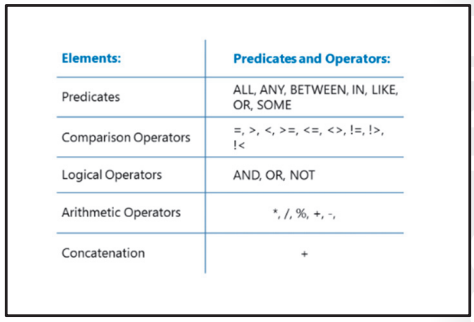
Like many programming languages, T-SQL contains elements that you will use in queries. You will use predicates to filter rows; operators to perform comparisons; functions and expressions to manipulate data or retrieve system information; and comments to document your code. If you need to go beyond writing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



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.

T-SQL Language Elements: Predicates and Operators

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



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

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: =.

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	~ (Bitwise NOT)
2	/, *, % (Division, Multiply, Modulo)
3	+, -, &, ^, | (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)

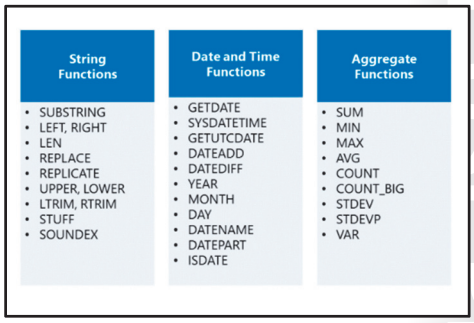
For more information on operator precedence, see Books Online at:

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
- 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 Books Online at:

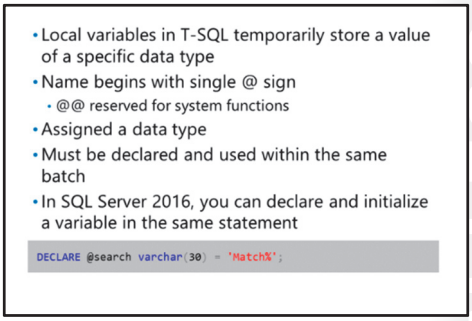
Built-in Functions

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.



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. For example:

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.

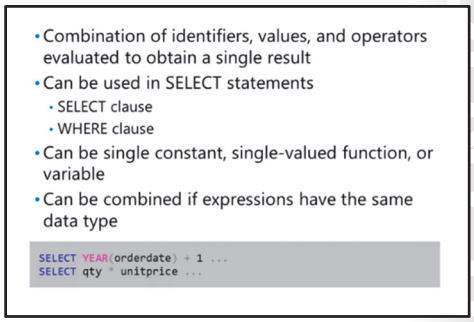
For more information on different types of tables, see:

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.

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:

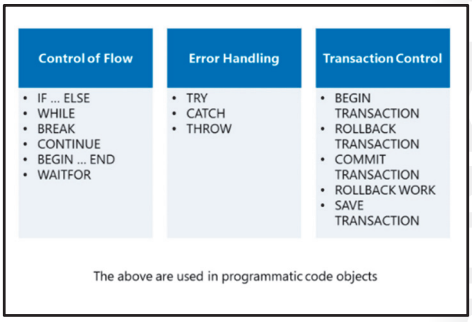
Expression

```
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.



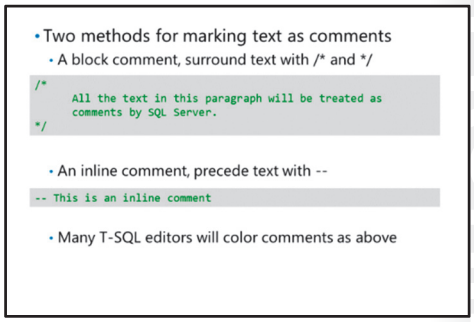
Some of the commonly-used control-of-flow statements include:

- **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.

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:

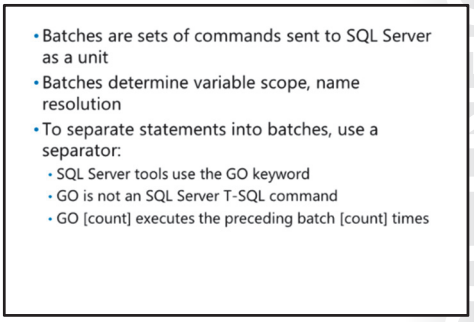
Block comment

```
/*
      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 than the surrounding T-SQL code. In SSMS, use the Tools, Options dialog box to customize the colors and fonts in the T-SQL script editor.

T-SQL Language Elements: Batch Separators

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



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

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

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

Code That Requires Multiple Batches

```
CREATE TABLE 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 work:

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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Open 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. On the **View** menu, click **Solution Explorer**.
9. In Solution Explorer, expand **Queries**, and then double-click the **11 – Demonstration A.sql** script file.
10. Select the code under **Step 1**, and then click **Execute**.
11. Select the code under **Step 2**, and then click **Execute**.
12. Select the code under **Step 3**, and then click **Execute**.
13. Select the code under **Step 4**, and then click **Execute**.
14. Select the code under **Step 5**, and then click **Execute**.
15. Select the code under **Step 6**, and then click **Execute**.
16. Select the code under **Step 7**, and then click **Execute**.
17. Select the code under **Step 8**, and then click **Execute**.
18. Select the code under the comment **Cleanup task if needed**, and then click **Execute**.
19. 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.

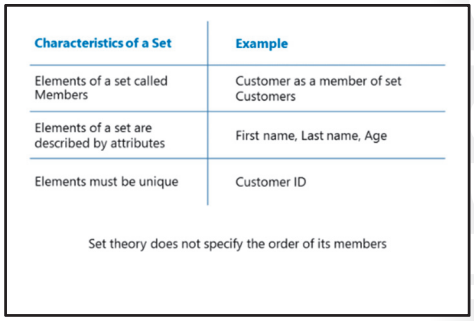
Lesson Objectives

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.



Characteristics of a Set	Example
Elements of a set called Members	Customer as a member of set 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

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, we 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.

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:

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

- 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.

 **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).

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	Carolos 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

Lesson 3
Understanding Predicate Logic

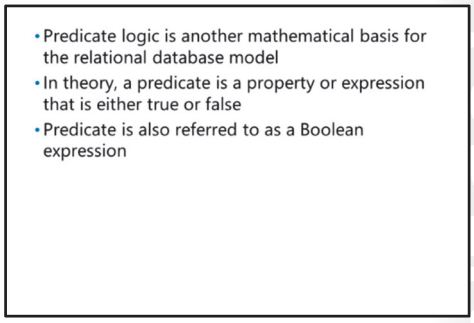
Along with set theory, predicate logic is a mathematical foundation for the relational database model, and with it, SQL Server 2016. You probably have a fair amount of experience with predicate logic—rather than the set theory—even if you have never used the term to describe it. This lesson will introduce 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



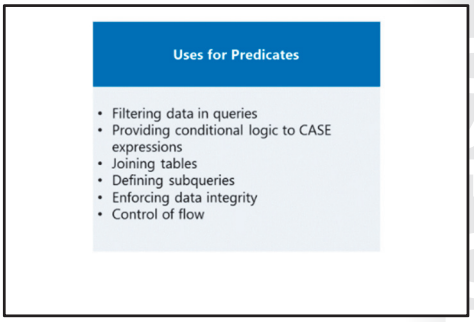
In theory, predicate logic is a framework for expressing logical tests that return true or false. A predicate is a property or expression that is true or false. You may have heard this referred to as a Boolean expression.

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 Applied to SQL Server Queries



As you have been learning, the ability to use predicates to express comparisons in terms of true, false, or unknown, is vital to writing effective queries in SQL Server. Although we have been discussing them separately, predicates do not stand alone, syntactically speaking. 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, we only need to consider values that are present, so predicates can result only in true or false values (known in predicate logic as "the law of the excluded middle"). In databases, however, you will likely have to account for missing values; 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

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, we need to look at a SELECT statement as a whole, including a number of optional elements. However, this lesson is not designed to provide detailed information about these elements—each part of a SELECT statement will be discussed in subsequent modules. Understanding the details of a WHERE clause, for example, is not required to recognize its place in the sequence of events.

Element	Expression	Role
SELECT	<select list>	Defines which columns to return
FROM	<table source>	Defines table(s) to query
WHERE	<search condition>	Filters 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

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, we 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.

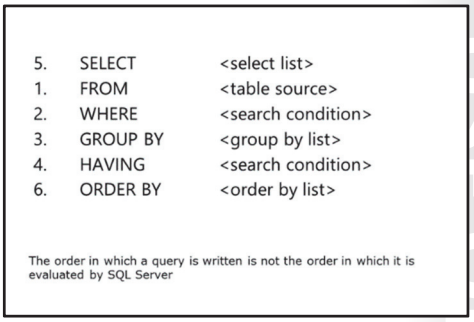
Not all elements will be present in every SELECT query. However, when an element is present, it will always be evaluated in the same order, with respect to the others present. For example, a WHERE clause will always be evaluated after the FROM clause and before a GROUP BY clause, if one exists.

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

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

Logical Query Processing

The order in which a SELECT statement is written is not that in which 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;
```

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

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

Change the database connection

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

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

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

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

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

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

 **Note:** Because the SELECT clause is evaluated after the other steps, any column aliases (in our example, OrderYear) created there cannot be used in the GROUP BY or HAVING.

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:

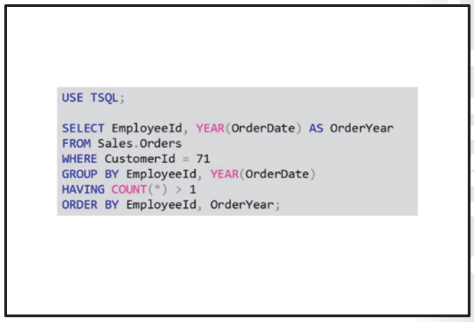
Logical Order

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

As we have seen, we 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 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.

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

5. In the **Login** box, type **Student**.
6. In the **Password** box, type **Pa$$w0rd**, and then click **Connect**.
7. If the **New Firewall Rule** dialog box appears, click **Sign In**, enter your Azure credentials, and then click **Sign in**. In the **New Firewall Rule** dialog box, ensure **Add my client IP** is selected and then click **OK**.
8. If the **Microsoft SQL Server Management Studio** dialog box appears, click **OK**.
9. On the **File** menu, point to **Open**, and then click **Project/Solution**.
10. In the **Open Project** dialog box, browse to the **D:\Demofiles\Mod02\Demo** folder, and then double-click **Demo.ssmssln**.
11. On the **View** menu, click **Solution Explorer**.
12. In Solution Explorer, double-click the **21 – Demonstration B.sql** script file.
13. In the **Available Databases** list, click **AdventureWorksLT**. If AdventureWorksLT is not in the list, right-click in the query pane, point to **Connection**, and then click **Change Connection**. Repeat steps 3 to 6 to connect to the server you created during preparation. Repeat this step to connect to **AdventureWorksLT**.
14. Select the code under the comment **Step 1**, and then click **Execute**.
15. Select the code under **Step 2**, and then click **Execute**.
16. Select the code under **Step 3**, and then click **Execute**.
17. Select the code under **Step 4**, and then click **Execute**.
18. Select the code under **Step 5**, and then click **Execute**.
19. Select the code under **Step 6**, and then click **Execute**.
20. Select the code under **Step 7**, and then click **Execute**.
21. Select the code under **Step 8**, and then click **Execute**.
22. 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

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: **20761A-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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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 script by clicking **Execute** on the toolbar (or press F5 on the keyboard). This will execute the whole script.
5. Observe the result and the database context.
6. Which database is selected in the Available Databases box?

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

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

1. Observe that, before the USE statement, there are the characters -- which means that the USE statement is treated as a comment. There is also a block comment around the whole T-SQL SELECT statement. Uncomment both statements.
2. First, execute the USE statement and then execute 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

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?

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	**3-25**

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.

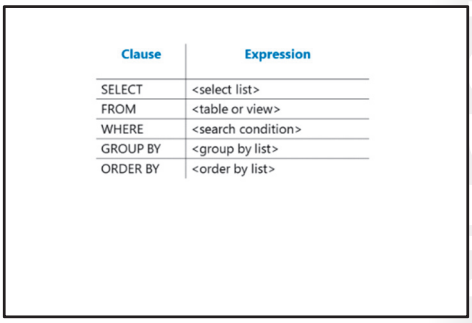
Lesson Objectives

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.



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

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.

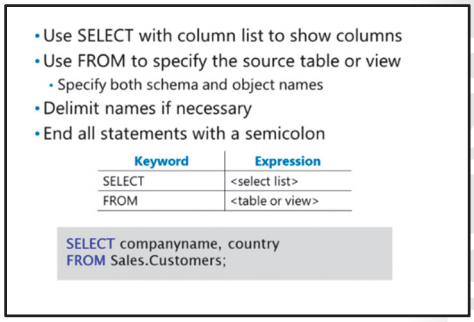
For more information on the SELECT elements, see:

SELECT (Transact-SQL)

http://aka.ms/vvwmme

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.



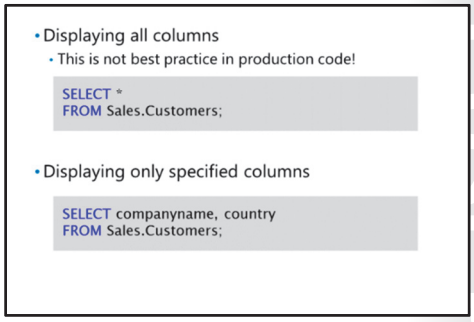
The FROM clause specifies the name of the table or view that is the source of the columns in the SELECT clause. To avoid errors in table or view name resolution, it is best to include the schema and object name, in the format SCHEMA.OBJECT—for example Sales.Customer.

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.

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.

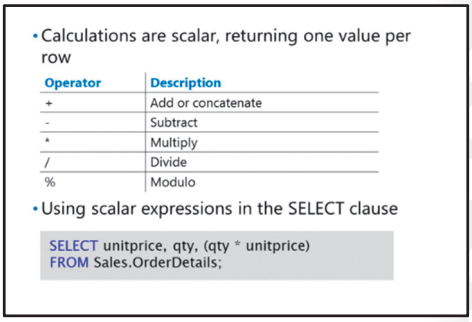


T-SQL supports the use of the asterisk, or "star" character (*) to substitute for an explicit column list. This will retrieve all columns from the source table. While the asterisk is suitable for a quick test, avoid using it in production work, as changes made to the table will cause the query to retrieve 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.

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.

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

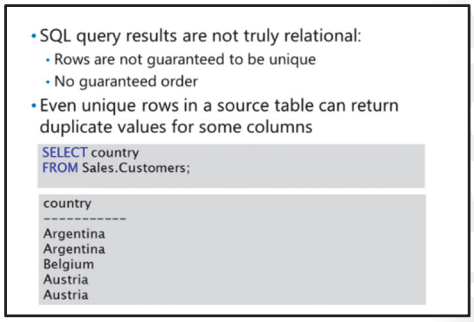
T-SQL queries may display duplicate rows, even if the source table has a key column enforcing uniqueness. Typically, this is the case when you retrieve only a few of the columns in a table. In this lesson, you will learn how to eliminate duplicates using the DISTINCT clause.

Lesson Objectives

In this lesson, you will learn to:

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

SQL Sets and Duplicate Rows



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

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

For example, consider a query that 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 prior to execution of SELECT clause

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

Continuing the previous example of countries from the Sales.Customers table, 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
```

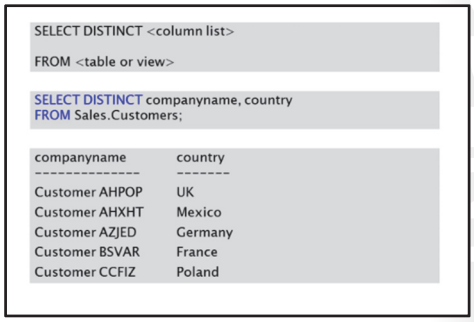
```
France
Germany
Ireland
Italy
Mexico
Norway
Poland
Portugal
Spain
Sweden
Switzerland
UK
USA
Venezuela
 (21 row(s) affected)
```

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

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.

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. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod03\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **Azure SQL** database engine instance using SQL Server authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod03\Demo folder.
3. 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.
4. In the **Available Databases** list, click **AdventureWorksLT**.
5. Select the code under the comment **Step 2**, and then click **Execute**.
6. Select the code under the comment **Step 3**, and then click **Execute**.
7. Select the code under the comment **Step 4**, and then click **Execute**.
8. 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.

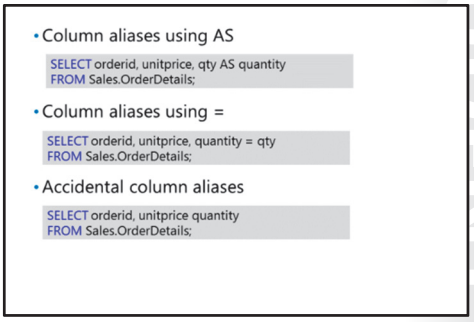
Lesson Objectives

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;
```

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?

SELECT Name AS ProductName FROM Production.Product

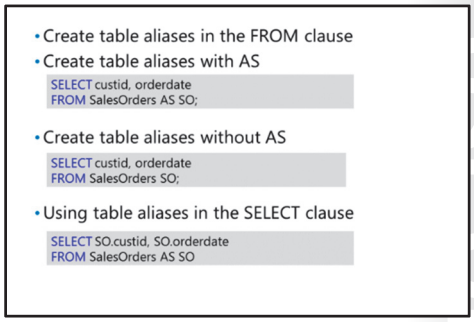
SELECT Name = ProductName FROM Production.Product

SELECT ProductName == Name FROM Production.Product

SELECT ProductName = Name FROM Production.Product

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:

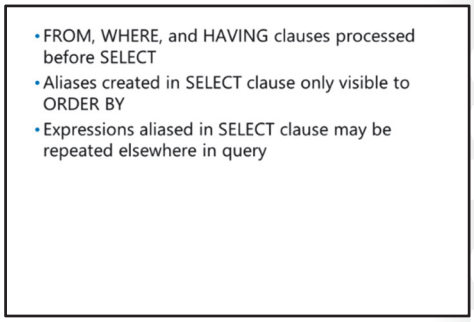
Table and Column Aliases Combined

```
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.

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:

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

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

Correct WHERE with Alias

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

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

The following statement will return an error:

Column Alias used in SELECT Clause

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

The resulting error:

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

Demonstration: Using Column and Table Aliases

In this demonstration, you will see how to:

- Use column and table aliases

Demonstration Steps

Use Column and Table Aliases

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod03\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **Azure SQL** database engine instance using SQL Server authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod03\Demo folder.
3. 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.
4. In the **Available Databases** list, click **AdventureWorksLT**.
5. Select the code under the comment **Step 2**, and then click **Execute**.
6. Select the code under the comment **Step 3**, and then click **Execute**.
7. Select the code under the comment **Step 4**, and then click **Execute**.
8. Select the code under the comment **Step 5**, and then click **Execute**.
9. 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 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.

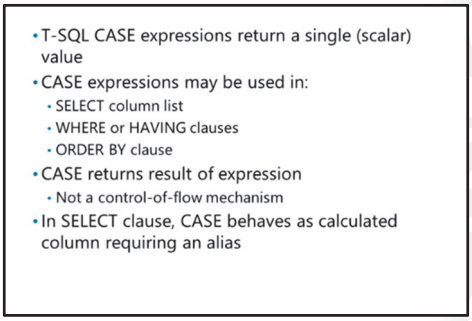
Lesson Objectives

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.



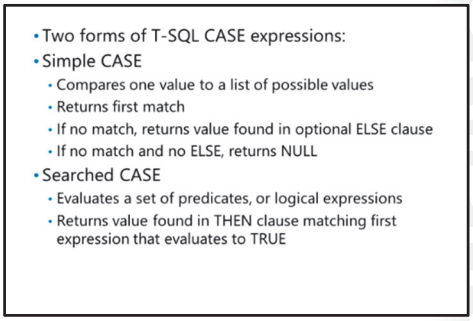
In T-SQL queries, CASE expressions are often used to provide an alternative value for one stored in the source table. For example, a CASE expression might be used to provide a friendly text name for something stored as a compact numeric code.

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.



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, just a simple substitution using a single table:

CASE Expression

```
SELECT productid, productname, categoryid,
        CASE categoryid
      WHEN 1 THEN 'Beverages'
      WHEN 2 THEN 'Condiments'
      WHEN 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 Books Online:

 **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. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod03\Setup.cmd as an administrator.

2. If SQL Server Management Studio is not already open, start it and connect to the **Azure SQL** database engine instance using SQL Server authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod03\Demo folder.
3. 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.
4. In the **Available Databases** list, click **AdventureWorksLT**.
5. Select the code under the comment **Step 2**, and then click **Execute**.
6. Select the code under the comment **Step 3**, and then click **Execute**.
7. 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: **20761A-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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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

 **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. Earlier, you learned how to write basic single-table queries. In this module, you will learn how to write queries that combine data from multiple sources in Microsoft® SQL Server®. You will 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

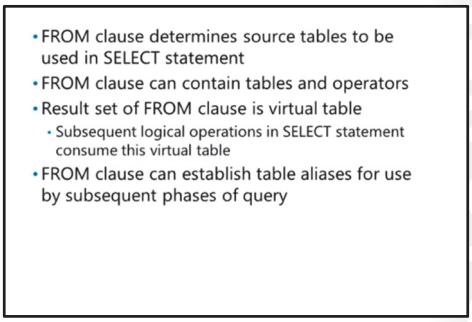
After completing this module, you will be able to:

- 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 multi-table queries.

The FROM Clause and Virtual Tables



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

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 learned earlier that the FROM clause is processed first, and as a result, any table aliases you create there may be referenced in the SELECT clause. You will see numerous examples of table aliases in this module. While 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:

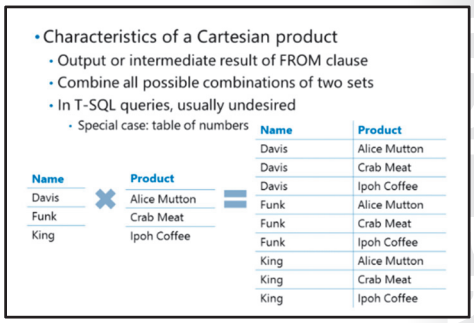
With Table Aliases

```
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 multi-table 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

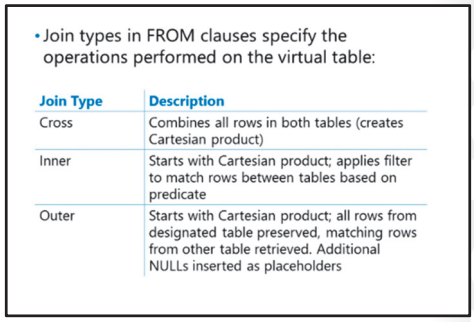


When learning about writing multi-table queries in T-SQL, it is important to understand the concept of Cartesian products. In mathematics, this is the product of two sets. The product of a set of two items and a set of six is a set of 12 items—or 6 x 2. In databases, a Cartesian product is the result of joining every row of one input table to all rows of another input table. The product of a table with 10 rows and a table with 100 rows is a result set with 1,000 rows. For most T-SQL queries, a Cartesian product is not the desired outcome. Typically, a Cartesian product occurs when two input tables are joined without considering any logical relationships between them. 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.

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:

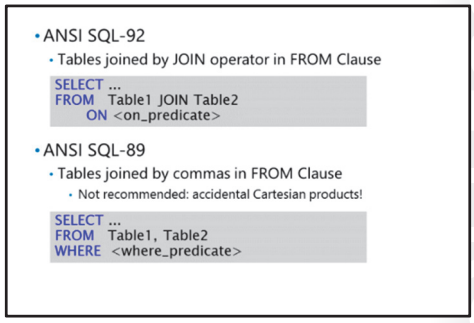


- A cross join operator (CROSS JOIN) adds all possible combinations of the two input tables' rows to the virtual table. Any filtering of the rows will happen in a WHERE clause. For most querying purposes, this operator is to be avoided.
- An inner join operator (INNER JOIN, or just JOIN) first creates 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.

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, as well as 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.



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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod04\Setup.cmd as an administrator. In the **User Account Control** dialog box, click **Yes**. When prompted, type **y**, and then press Enter. When the script has completed, press any key.
3. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod04\Demo folder.

5. If Solution Explorer is not visible, on the **View** menu, click **Solution Explorer**.
6. In Solution Explorer, expand **Queries**, and then double-click the **11 – Demonstration A.sql** script file.
7. Select the code under the comment **Step 1**, and then click **Execute**.
8. Select the code under the comment **Step 2**, and then click **Execute**.
9. Select the code under the comment **Step 3**, and then click **Execute**.
10. Select the code under the comment **Step 4**, and then click **Execute**.
11. Select the code under the comment **Step 5**, and then click **Execute**.

Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Question

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

Select the correct answer.

	A cross join
	An inner join
	An outer left join
	An outer right join

Lesson 2
Querying with Inner Joins

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

Lesson Objectives

After completing this lesson, you will be able to:

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

Understanding Inner Joins

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

- 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

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

SQL-89 and SQL-92 Join Syntax Compared

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

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

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

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

ANSI-92 Join

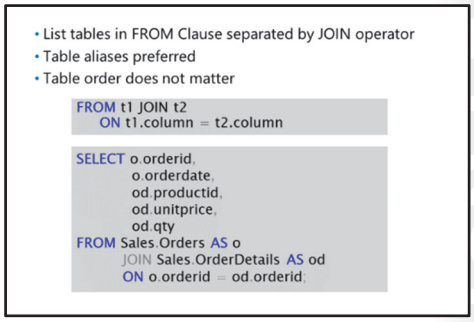
```
1)    SELECT c.companyname, o.orderdate
2)    FROM Sales.Customers AS c
3)    JOIN Sales.Orders AS o
4)    ON c.custid = o.custid;
```

As you learned earlier, the FROM clause will be processed before the SELECT clause. 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:

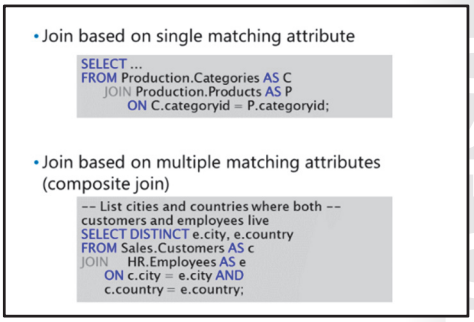


- As you have seen, table aliases are preferred, not only for the SELECT list, but also for expressing the ON clause.
- Inner joins may be performed on a single matching attribute, such as an orderid, or on multiple matching attributes, such as the combination of orderid and productid. Joins that match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

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. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod04\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod04\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Select the code under the comment **Step 1**, and then click **Execute**.
5. Select the code under the comment **Step 2**, and then click **Execute**.
6. Select the code under the comment **Step 3**, and then click **Execute**.
7. Select the code under the comment **Step 4**, and then click **Execute**.
8. Select the code under the comment **Step 5**, and then click **Execute**.
9. Select the code under the comment **Step 6**, and then click **Execute**.
10. 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 multi-table query can provide an alternative view of your business data. As with inner joins, you will express a logical relationship between the tables. However, you will retrieve not only rows with matching attributes, but also all rows present in one of the tables, whether or not there is a match in the other table.

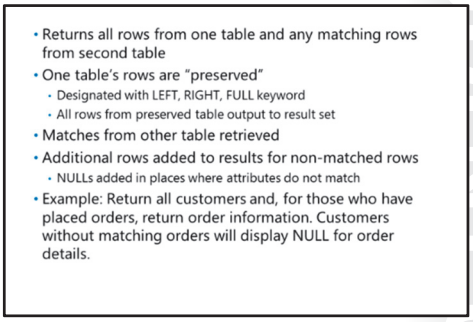
Lesson Objectives

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 non-matching rows.

Understanding Outer Joins

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



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.

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

Outer Left Join

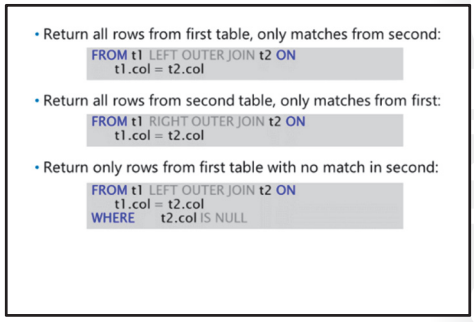
```
USE AdventureWorks;
GO

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:

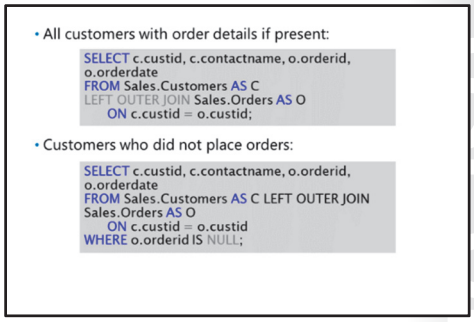


- As you have seen, table aliases are preferred not only for the SELECT list, but also for expressing the ON clause.
- Outer joins are expressed using the keywords LEFT, RIGHT, or FULL preceding OUTER JOIN. The purpose of the keyword is to indicate which table (on which side of the keyword JOIN) should 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.
- Multi-table joins are more complex when an OUTER JOIN is present. The presence of NULLs in the results of an outer join may cause issues if the intermediate results are then joined, via an inner join, to a third table. Rows with NULLs may be filtered out by the second join's predicate.
- To display only rows where no match exists, add a test for NULL in a WHERE clause following an OUTER JOIN predicate.

Outer Join Examples

The following are some examples of outer joins:

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



Outer Join Example

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

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

Outer Join Example

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

Demonstration: Querying with Outer Joins

In this demonstration, you will see how to:

- Use outer joins.

Demonstration Steps

Use Outer Joins

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod04\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod04\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. Select the code under the comment **Step 1**, and then click **Execute**.
5. Select the code under the comment **Step 2**, and then click **Execute**.
6. Select the code under the comment **Step 3**, and then click **Execute**.
7. Select the code under the comment **Step 4**, and then click **Execute**.
8. Select the code under the comment **Step 5**, and then click **Execute**.
9. Select the code under the comment **Step 6**, and then click **Execute**.
10. Select the code under the comment **Step 7**, and then click **Execute**.
11. 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.

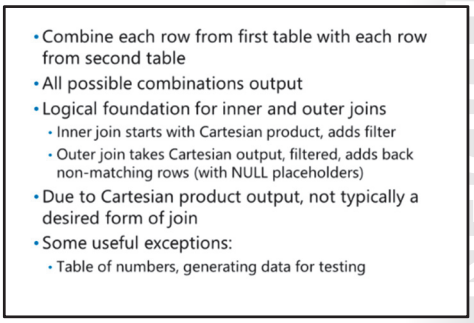
Lesson Objectives

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;
```

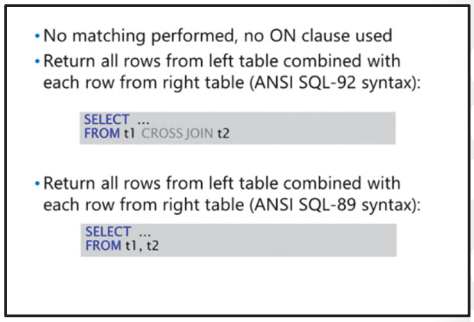
While this is not typically a desired output, there are a few practical applications for writing an explicit cross join:

- 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.

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.



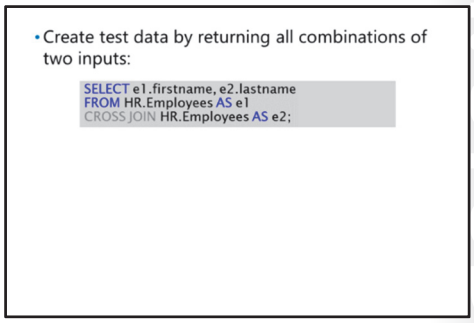
Cross Join Examples

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

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

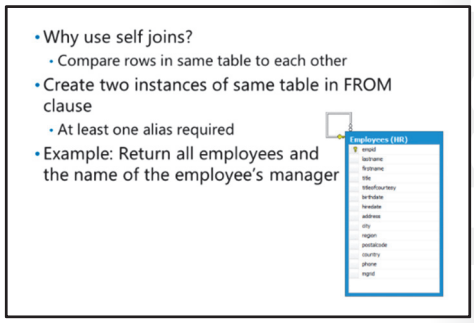
Cross Join Example

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



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.



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;
```

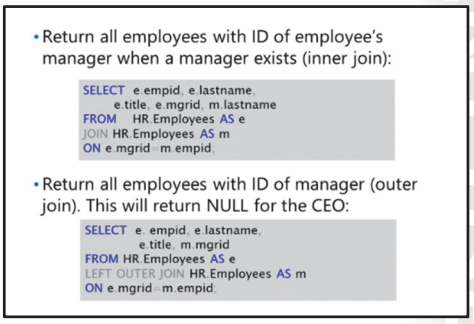
This yields results like the following:

```
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:



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. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod04\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod04\Demo folder.
3. In Solution Explorer, open the **41 – Demonstration D.sql** script file.
4. Select the code under the comment **Step 1**, and then click **Execute**.
5. Select the code under the comment **Step 2**, and then click **Execute**.
6. Select the code under the comment **Step 3**, and then click **Execute**.
7. Select the code under the comment **Step 4**, and then click **Execute**.

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: **20761A-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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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 project file D:\Labfiles\Lab04\Starter\Project\Project.ssmssln and the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. The developer has written this query:

```
SELECT
custid, contactname, orderid
FROM Sales.Customers
INNER join Sales.Orders ON Customers.custid = Orders.custid;
```

   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 project file **D:\Labfiles\Lab04\Starter\Project\Project.ssmssln** and the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the TSQL database.
2. To better understand the needed tasks, you will first write a SELECT statement against the HR.Employees table showing the empid, lastname, firstname, title, and mgrid columns.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\72 - Lab Exercise 3 - Task 1 Result.txt. Notice the values in the mgrid column. The mgrid column is in a relationship with empid column. This is called a self-referencing relationship.

▶ Task 2: Write a Query That Uses a Self Join

1. Copy the SELECT statement from task 1 and modify it to include additional columns for the manager information (lastname, firstname) using a self join. Assign the aliases mgrlastname and mgrfirstname respectively, to distinguish the manager names from the employee names.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\73 - Lab Exercise 3 - Task 2 Result.txt. Notice the number of rows returned.
3. Is it mandatory to use table aliases when writing a statement with a self join? Can you use a full source table name as an alias? Please explain.
4. Why did you get fewer rows in the T-SQL statement under task 2 compared to task 1?

Results: After this exercise, you should have an understanding of how to write T-SQL statements that use self joins.

Exercise 4: Writing Queries That Use Outer Joins

Scenario

The sales department was satisfied with the report you produced in exercise 2. Now sales staff would like to change the report to show all customers, even if they did not have any orders, and still include order information for the customers who did. You need to write a SELECT statement to retrieve all rows from Sales.Customers (columns custid and contactname) and the orderid column from the table Sales.Orders.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement That Uses an Outer Join

▶ Task 1: Write a SELECT Statement That Uses an Outer Join

1. Open the project file **D:\Labfiles\Lab04\Starter\Project\Project.ssmssln** and the T-SQL script **81 - Lab Exercise 4.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the custid and contactname columns from the Sales.Customers table and the orderid column from the Sales.Orders table. The statement should retrieve all rows from the Sales.Customers table.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.
4. Notice the values in the column orderid. Are there any missing values (marked as NULL)? Why?

Results: After this exercise, you should have a basic understanding of how to write T-SQL statements that use outer joins.

Exercise 5: Writing Queries That Use Cross Joins

Scenario

The HR department would like to prepare a personalized calendar for each employee. The IT department supplied you with T-SQL code that will generate a table with all dates for the current year. Your job is to write a SELECT statement that would return all rows in this new calendar date table for each row in the HR.Employees table.

The main tasks for this exercise are as follows:

1. Execute the T-SQL Statement
2. Write a SELECT Statement That Uses a Cross Join
3. Drop the HR.Calendar Table

▶ Task 1: Execute the T-SQL Statement

1. Open the project file **D:\Labfiles\Lab04\Starter\Project\Project.ssmssln** and the T-SQL script **91 - Lab Exercise 5.sql**. Ensure that you are connected to the TSQL database.
2. Execute the T-SQL code under task 1. Don't worry if you do not understand the provided T-SQL code, as it is used here to 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

1. Execute the provided T-SQL statement to remove the HR.Calendar table.

Results: After this exercise, you should have an understanding of how to write T-SQL statements that use cross joins.

Module Review and Takeaways

Best Practice: Table aliases should always be defined when joining tables. Joins should be expressed using SQL-92 syntax, with JOIN and ON keywords.

Review Question(s)

Question: How does an inner join differ from an outer join?

Question: Which join types include a logical Cartesian product?

Question: Can a table be joined to itself?

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.

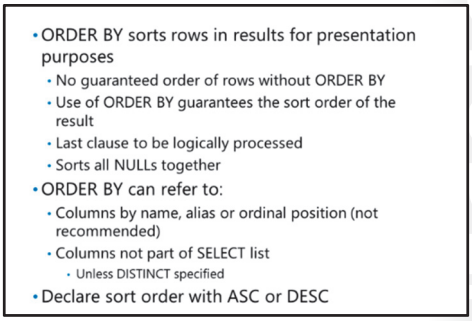
Lesson Objectives

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 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 non-unique values in the first column.

Column aliases. Remember that ORDER BY is processed after the SELECT clause and therefore has access to aliases defined in the SELECT list.

Columns by position in the SELECT clause. This is not recommended, 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 hire to least recent, with employees hired on the same date listed alphabetically by last name:

Ascending and Descending Sort

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

For additional documentation on the ORDER BY clause, see Books Online at:

ORDER BY Clause (Transact-SQL)

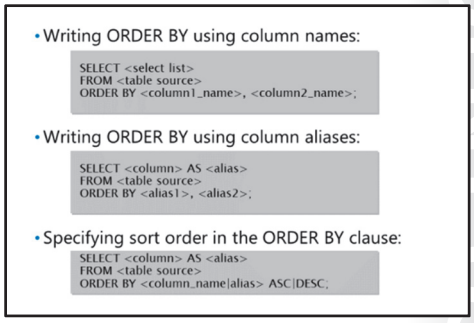
http://go.microsoft.com/fwlink/?LinkID=402718

ORDER BY Clause Syntax

The syntax of the ORDER BY clause appears as follows:

ORDER BY Clause

```
ORDER BY <order_by_list>
OFFSET <offset_value> ROW|ROWS FETCH
FIRST|NEXT <fetch_value> ROW|ROWS ONLY
```



Note: The use of the OFFSET-FETCH option in the ORDER BY clause will be covered later in this module.

Most variations of ORDER BY will occur in the ORDER BY list. To specify columns by name, with the default ascending order, use the following syntax:

ORDER BY List

```
ORDER BY <column_name_1>, <column_name_2>;
```

A fragment of code using columns from the Sales.Customers table would look like this:

ORDER BY List Example

```
ORDER BY country, region, city;
```

To specify columns by aliases defined in the SELECT clause, use the following syntax:

ORDER BY an Alias

```
SELECT <column_name_1> AS alias1, <column_name_2> AS alias2
FROM <table source>
ORDER BY alias1;
```

A query for the Sales.Orders table using column aliases would look like this:

ORDER BY Using Column Alias Example

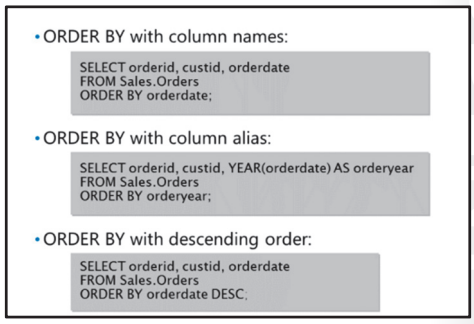
```
SELECT orderid, custid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
ORDER BY orderyear;
```

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 Example 1

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

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

ORDER BY Example 2

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

A query against the HR.Employees table, which sorts employees in descending order of hire date (that is, most recent to oldest), using lastname to differentiate employees hired on the same date:

ORER 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

1. Ensure that the MSL-TMG1, 20761A-MIA-DC, and 20761A-MIA-SQL virtual machines are running, and then log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Start SQL Server Management Studio and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication.
3. If the **Microsoft SQL Server Management Studio** dialog box appears, click **OK**.
4. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod05\Demo folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Expand **Queries**, and double-click the **11 – Demonstration A.sql** script file.
7. In the Available Databases list, click ADVENTUREWORKSLT.
8. Select the code under the comment **Step 1**, and then click **Execute**.
9. Select the code under **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: 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?

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.

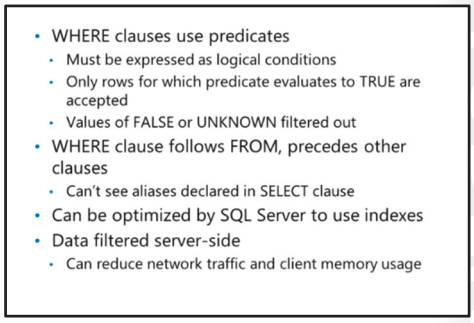
Lesson Objectives

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.

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 Books Online, the syntax of the WHERE clause appears as follows:

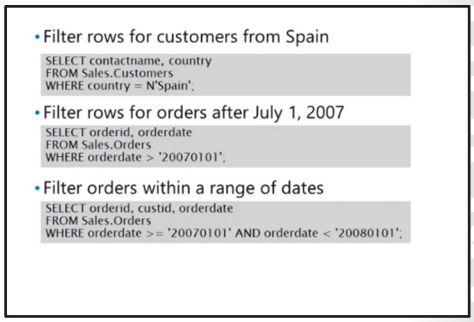
WHERE Clause Syntax

```
WHERE <search_condition>
```

The most common form of a WHERE clause is as follows:

Typical WHERE Clause

```
WHERE <column> <operator> <expression>
```



For example, the following code fragment shows a WHERE clause that will filter only customers from Spain:

WHERE Clause Example

```
SELECT contactname, country
FROM Sales.Customers
WHERE country = N'Spain';
```

Any of the logical operators introduced in the T-SQL language module earlier in this course may be used in a WHERE clause predicate.

This example filters orders placed after a specified date.

WHERE Clause Example

```
SELECT orderid, orderdate
FROM Sales.Orders
WHERE orderdate > '20070101';
```

Note: The representation of dates as strings delimited by quotation marks will be covered in the next module.

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 the next module.

Demonstration: Filtering Data with Predicates

In this demonstration, you will see how to:

- Filter data in a WHERE clause

Demonstration Steps

Filter Data in a WHERE Clause

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the MSL-TMG1, 20761A-MIA-DC, and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. If SQL Server Management Studio is not already open, start it and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod05\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. In the **Available Databases** list, click **ADVENTUREWORKSLT**.
5. Select the code under the comment **Step 1**, and then click **Execute**.
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**.
8. Select the code under the comment **Step 4**, and then click **Execute**.
9. Select the code under the comment **Step 5**, and then click **Execute**.
10. Select the code under the comment **Step 6**, and then click **Execute**.
11. Select the code under the comment **Step 7**, and then click **Execute**.
12. Select the code under the comment **Step 8**, and then click **Execute**.
13. Select the code under the comment **Step 9**, and then click **Execute**.
14. Select the code under the comment **Step 10**, and then click **Execute**.
15. Select the code under the comment **Step 11**, and then click **Execute**.
16. 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.

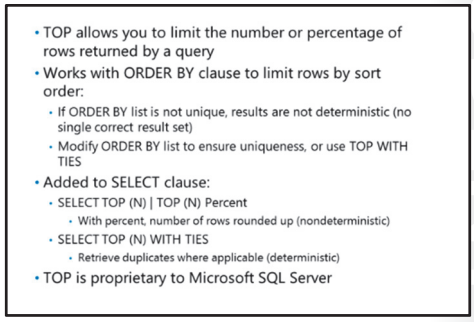
Lesson Objectives

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

For additional information about the TOP clause, see Books Online at:

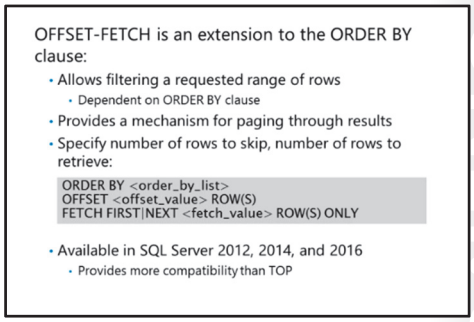
TOP (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402719

Filtering in the ORDER BY Clause Using OFFSET-FETCH

While the TOP option is used by many SQL Server professionals as a method for retrieving only a certain range of rows, it also has 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.

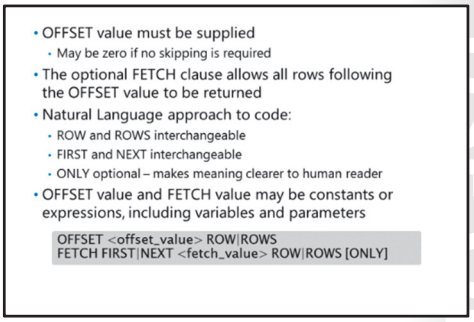
For more information about the OFFSET-FETCH clause, see *Using OFFSET and FETCH to limit the rows returned* in Books Online at:

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]
```

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 may 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;
```

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. Ensure that you have completed the previous demonstration in this module. Alternatively, start the MSL-TMG1, 20761A-MIA-DC, and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd.**
2. If SQL Server Management Studio is not already open, start it and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod05\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. In the **Available Databases** list, ensure **ADVENTUREWORKSLT** is selected.
5. Select the code under the comment **Step 1**, and then click **Execute**.
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**.
8. Select the code under the comment **Step 4**, and then click **Execute**.
9. Select the code under the comment **Step 5**, and then click **Execute**.
10. Select the code under the comment **Step 6**, and then click **Execute**.
11. Select the code under the comment **Step 7**, and then click **Execute**.
12. Select the code under the comment **Step 8**, and then click **Execute**.
13. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Question	
You have a table named Products in your Sales database. You are creating a paged display of products in an application that shows 20 products on each page, ordered by name. Which of the following queries would return the third page of products?	
Select the correct answer.	
	SELECT ProductID, ProductName, ProductNumber FROM Sales.Products ORDER BY ProductName ASC OFFSET 60 ROWS FETCH NEXT 20 ROWS ONLY
	SELECT ProductID, ProductName, ProductNumber FROM Sales.Products ORDER BY ProductName ASC OFFSET 40 ROWS FETCH NEXT 20 ROWS ONLY;
	SELECT TOP (20) ProductID, ProductName, ProductNumber FROM Sales.Products ORDER BY ProductName ASC
	SELECT TOP (20) WITH TIES ProductID, ProductName, ProductNumber FROM Sales.Products ORDER BY ProductName ASC

Lesson 4
Working with Unknown Values

Unlike traditional Boolean logic, predicate logic in SQL Server needs to account for missing values and deal with cases where the result of a predicate is unknown. In this lesson, you will learn how three-valued logic accounts for unknown and missing values; how SQL Server uses NULL to mark missing values; and how to test for NULL in your queries.

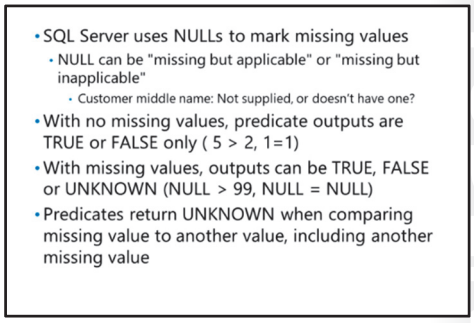
Lesson Objectives

After completing this lesson, you will be able to:

- Describe three-valued logic.
- Describe the use of NULL in queries.

Three-Valued Logic

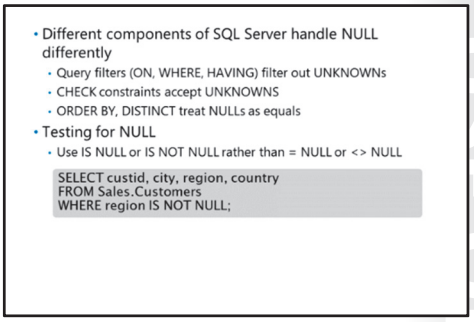
Earlier in this course, you learned that SQL Server uses predicate logic as a framework for logical tests that return TRUE or FALSE. This is true for logical expressions where all values being tested are present. If you know the values of both X and Y, you can safely determine whether X>Y is TRUE or FALSE.



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.

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.

Note the output of the following queries:

ORDER BY Query that includes NULL in Results

```
SELECT empid, lastname, region
FROM HR.Employees
ORDER BY region ASC; --Ascending sort order explicitly included for clarity.
```

This returns the following, with all employees whose region is missing (marked as NULL) sorted first:

```
empid       lastname             region
----------- -------------------- ---------------
5           Buck                 NULL
6           Suurs                NULL
7           King                 NULL
9           Dolgopyatova         NULL
8           Cameron              WA
1           Davis                WA
2           Funk                 WA
3           Lew                  WA
4           Peled                WA
```

Note: A common question about controlling the display of NULL in queries is whether NULLs can be forced to the end of a result set. As you can see, the ORDER BY clause sorts the NULLs together and first—a behavior you cannot override.

- ORDER BY treats NULLs as if they were the same value and always sorts NULLs together, putting them first in a column. Make sure you test the results of any queries in which the column being used for sort order contains NULLs, and understand the impact of ascending and descending sorts on NULLs.
- In ANSI-compliant queries, a NULL is never equivalent to another value, even another NULL. Queries written to test NULL with an equality will fail to return correct results.

Note the following example:

Incorrectly Testing for NULL

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

This returns unexpected results:

```
empid       lastname             region
----------- -------------------- ---------------
 (0 row(s) affected)
```

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

See the following example:

Correctly Testing for NUL

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

This returns correct results:

```
empid       lastname             region
----------- -------------------- ---------------
5           Buck                 NULL
6           Suurs                NULL
7           King                 NULL
9           Dolgopyatova         NULL
 (4 row(s) affected)
```

Demonstration: Working with NULL

In this demonstration, you will see how to:

- Test for NULL

Demonstration Steps

Test for Null

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the MSL-TMG1, 20761A-MIA-DC, and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. If SQL Server Management Studio is not already open, start it and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod05\Demo folder.
3. In Solution Explorer, open the **41 – Demonstration D.sql** script file.
4. In the **Available Databases** list, ensure **ADVENTUREWORKSLT** is selected.
5. Select the code under the comment **Step 2**, and then click **Execute**.
6. Select the code under the comment **Step 3**, and then click **Execute**.
7. Select the code under the comment **Step 4**, and then click **Execute**.
8. Select the code under the comment **Step 5**, and then click **Execute**.
9. Select the code under the comment **Step 6**, and then click **Execute**.
10. Select the code under the comment **Step 7**, and then click **Execute**.
11. 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: **20761A-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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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 project file **D:\Labfiles\Lab05\Starter\Project\Project.ssmssln** and the T-SQL script **61 - Lab Exercise 2.sql**. Ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement to retrieve the **custid** and **contactname** columns from the **Sales.Customers** table and the **orderid** and **orderdate** columns from the **Sales.Orders** table. Filter the results to include only orders placed on or after April 1, 2008 (filter the **orderdate** column), then sort the result by orderdate in descending order and custid in ascending order.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file **D:\Labfiles\Lab05\Solution\62 - Lab Exercise 2 - Task 1 Result.txt**.

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

1. Someone took your T-SQL statement from lab 4 and added the following WHERE clause:

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid,
m.lastname AS mgrlastname, m.firstname AS mgrfirstname
FROM HR.Employees AS e
INNER JOIN HR.Employees AS m ON e.mgrid = m.empid
WHERE mgrlastname = 'Buck';
```

2. Execute the query exactly as written inside a query window and observe the result.
3. 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 project file D:\Labfiles\Lab05\Starter\Project\Project.ssmssln and the T-SQL script 71 - Lab Exercise 3.sql. Ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement against the **Sales.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 project file **D:\Labfiles\Lab05\Starter\Project\Project.ssmssln** and the T-SQL script **81 - Lab Exercise 4.sql.** Ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement to retrieve the **custid**, **orderid**, and **orderdate** columns from the **Sales.Orders** table. Order the rows by orderdate and ordered, and then retrieve the first 20 rows.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file **D:\Labfiles\Lab05\Solution\82 - Lab Exercise 4 - Task 1 Result.txt**.

▶ Task 2: Use the OFFSET-FETCH Clause to Skip the First 20 Rows

1. Copy the SELECT statement in task 1 and modify the OFFSET-FETCH clause to skip the first 20 rows and fetch the next 20.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file **D:\Labfiles\Lab05\Solution\83 - Lab Exercise 4 - Task 2 Result.txt**.

▶ Task 3: Write a Generic Form of the OFFSET-FETCH Clause for Paging

- 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

Review Question(s)

Question: Does the physical order of rows in an SQL Server table guarantee any sort order in queries using the table?

Question: You have the following query:

```
SELECT p.PartNumber, p.ProductName, o.Quantity
FROM Sales.Products AS p
LEFT OUTER JOIN Sales.OrderItems AS o
ON p.ID = o.ProductID
ORDER BY o.Quantity ASC
```

You have one new product that has yet to receive any orders. Will this product appear at the top or the bottom of the results?

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 **6-21**

Lab: Working with SQL Server 2016 Data Types **6-27**

Module Review and Takeaways **6-33**

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 which 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 may 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 may 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.

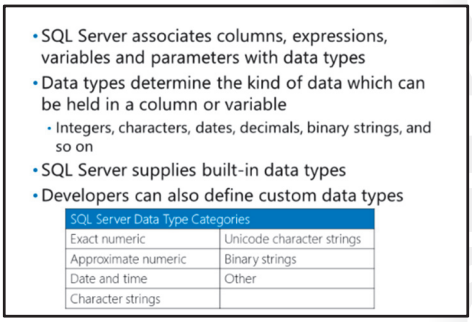
Lesson Objectives

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 may also extend the supplied set by creating aliases to built-in types and even by producing new user-defined types using the Microsoft .NET Framework; however, this lesson will focus on the built-in system data types.

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.

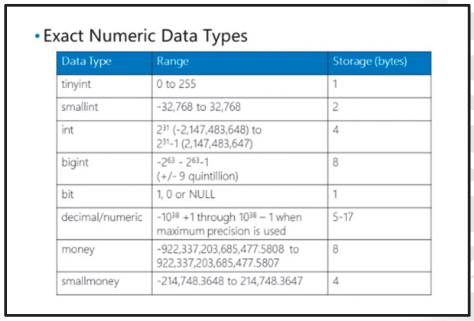
  - Decimals – decimal numbers with control over both the total number of digits stored and the number of digits to the right of the decimal place.

- **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 which 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.



Data Type	Range	Storage (bytes)
tinyint	0 to 255	1
smallint	-32,768 to 32,768	2
int	2^{31} (-2,147,483,648) to $2^{31}-1$ (2,147,483,647)	4
bigint	-2^{63} - $2^{63}-1$ (+/- 9 quintillion)	8
bit	1, 0 or NULL	1
decimal/numeric	$-10^{38}+1$ through $10^{38}-1$ when maximum precision is used	5-17
money	-922,337,203,685,477.5808 to 922,337,203,685,477.5807	8
smallmoney	-214,748.3648 to 214,748.3647	4

  - 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.
  - 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 Books Online:

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*.

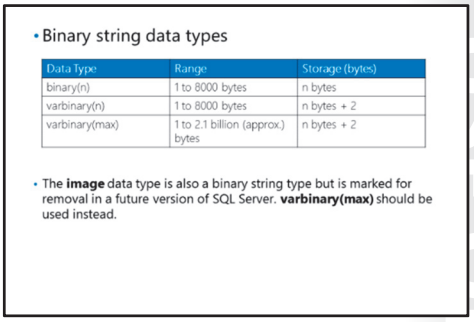
See the topic *float and real (Transact-SQL)* in Books Online at:

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 Books Online at:

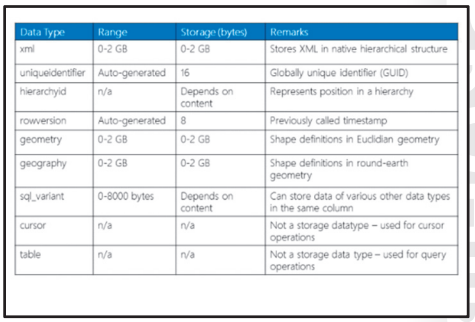
binary and varbinary (Transact-SQL)

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:



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

- 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.

 **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 various methods 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 which 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 coordinated. 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 which 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.

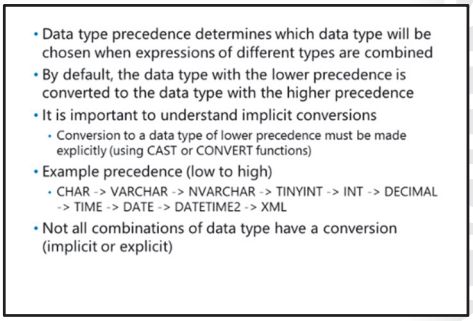
For information on all of SQL Server's data types, see Books Online, starting from:

Data Types (Transact-SQL)

http://aka.ms/we8bzv

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

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 Books Online at:

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)*:

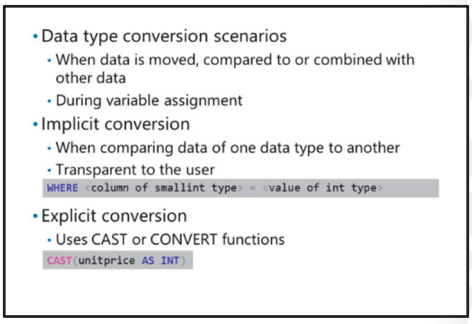
CAST and CONVERT (Transact-SQL) – Implicit Conversions

http://aka.ms/asaqq3

When are Data Types Converted?

When querying SQL Server, there is 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.



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 Books Online at:

Data Type Conversion (Database Engine)

http://aka.ms/t5db1i

Demonstration: SQL Server Data Types

In this demonstration, you will see how to:

- Convert data types

Demonstration Steps

Convert Data Types

1. Ensure that the 20761A-MIA-DC, and 20761A-MIA-SQL virtual machines are running, and then log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Start SQL Server Management Studio and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication.
3. If the **New Firewall Rule** dialog box appears, click **Sign In**, enter your Azure credentials, and then click **Sign in**. In the **New Firewall Rule** dialog box, ensure **Add my client IP** is selected, and then click **OK**.
4. If the **Microsoft SQL Server Management Studio** dialog box appears, click **OK**.

5. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod06\Demo folder.
6. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
7. Expand **Queries**, and double-click the **11 – Demonstration A.sql** script file.
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. 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 may find useful in your queries.

Lesson Objectives

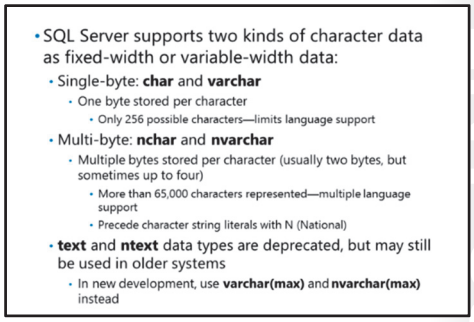
After completing this lesson, you will be able to:

- 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.

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 which 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) **nchar**(n)	1-8000 characters 1-4000 characters	n bytes, padded 2*n bytes, padded
varchar(n) **nvarchar**(n)	1-8000 characters 1-4000 characters	Actual length + 2 bytes
varchar(**max**) **nvarchar**(**max**)	Up to 2 GB	Actual length + 2 bytes

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

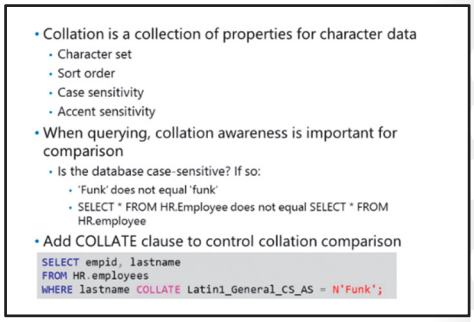
- Single-byte character data is indicated with single quotation marks alone—for example 'SQL Server'.
- Multi-byte character data is indicated by single quotation marks with the prefix N (for National)— for example N'SQL Server'. The N prefix is always required, even when inserting the data into a column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

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 Books Online:

COLLATE (Transact-SQL)

http://aka.ms/ty97q8

Collation and Unicode Support

http://aka.ms/pm56d9

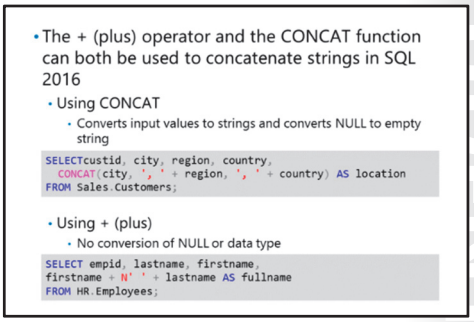
String Concatenation

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 Books Online at:

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 +

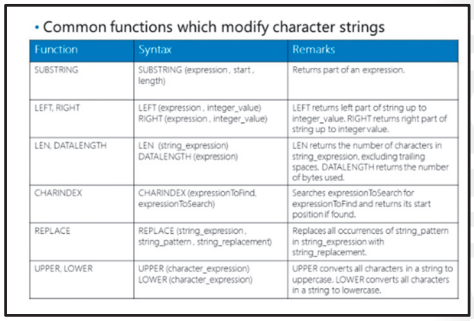
```
SELECT
empid, lastname, firstname, firstname + N' ' + lastname AS fullname
FROM HR.Employees;
```

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.



• Common functions which 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.

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;
```

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
```

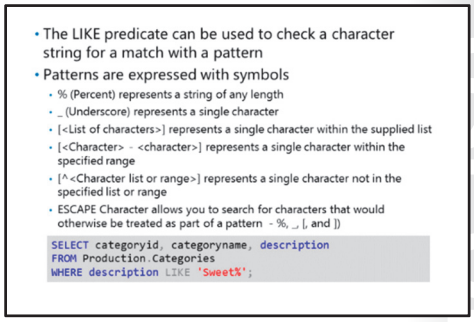
For references on these and other string functions, see Books Online at:

String Functions (Transact-SQL)

http://aka.ms/lt6hg9

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, allowing you to 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).

For further information on LIKE, see Books Online at:

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. Ensure that you have completed the previous demonstration in this module. Alternatively, start the MSL-TMG1, 20761A-MIA-DC, and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. If SQL Server Management Studio is not already open, start it and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod06\Demo folder.

3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. In the **Available Databases** list, click **AdventureWorksLT**.
5. Select the code under the comment **Step 2**, and then click **Execute**.
6. Select the code under the comment **Step 3**, and then click **Execute**.
7. Select the code under the comment **NOTE: this will return no results**, and then click **Execute**.
8. Select the code under the comment **Step 4**, and then click **Execute**.
9. Select the code under the comment **Step 5**, and then click **Execute**.
10. Select the code under the comment **Step 6**, and then click **Execute**.
11. Select the code under the comment **end FORMAT example**, and then click **Execute**.
12. Select the code under the comment **Step 7**, and then click **Execute**.
13. 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 in 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.

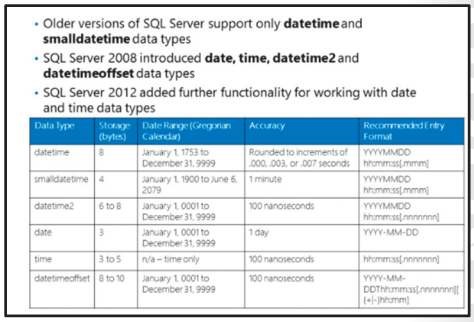
Lesson Objectives

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).

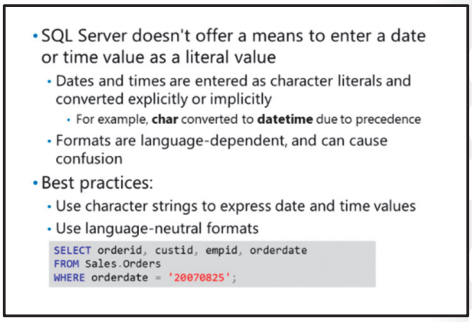
For more information on all the date and time data types, see Books Online at:

Date and Time Types

http://aka.ms/aekgy8

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 may also explicitly convert string literals with the T-SQL CAST and CONVERT functions, which you will learn about in the next module.)



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

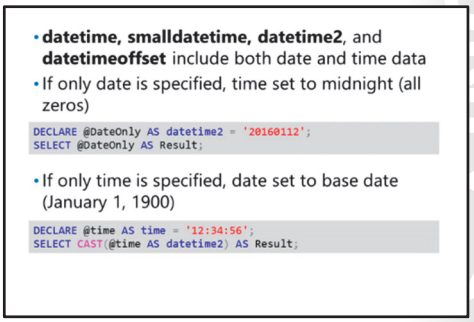
```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate = '20070825';
```

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.nnnnnnn' 'YYYY-MM-DD hh:mm:ss.nnnnnnn' 'YYYY-MM-DDThh:mm:ss.nnnnnnn' 'YYYYMMDD' 'YYYY-MM-DD'	'2014-02-12' '20140212 12:30:15.1234567' '2014-02-12 12:30:15.1234567' '2014-02-12T12:30:15.1234567' '20140212' '2014-02-12'
date	'YYYYMMDD' 'YYYY-MM-DD'	'20140212' '2014-02-12'
time	'hh:mm:ss.nnnnnnn'	'12:30:15.1234567'
datetimeoffset	'YYYYMMDD hh:mm:ss.nnnnnnn [+|-]hh:mm' 'YYYY-MM-DD hh:mm:ss.nnnnnnn	'20140212 12:30:15.1234567 +02:00' '2014-02-12 12:30:15.1234567 +02:00'

Data Type	Language-Neutral Formats	Examples
	[+|-]hh:mm' 'YYYYMMDD' 'YYYY-MM-DD'	'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 a 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 which 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;
```

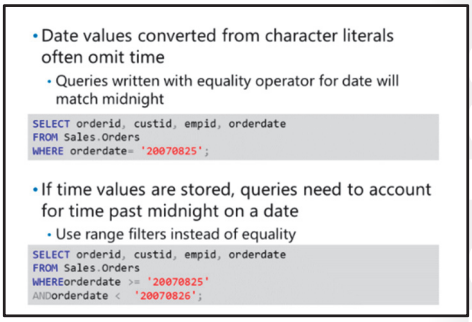
Returns:

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

Querying Date and Time Values

When querying date and time data types, you may 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:



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 which 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.

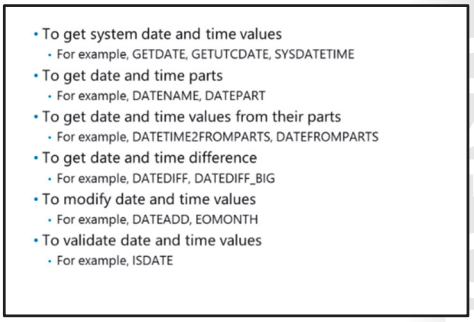
One way to be certain of returning all the orders for August 25, 2007—regardless of the time portion of the orderdate column—would be to query the data with a range, rather than a single value:

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, as well as whether to include or exclude time zone information.
- Functions that return parts of date and time values, enabling you to extract only the portion of a date or time that your query requires. Note that DATENAME and DATEPART offer functionality similar to one another. The difference between them is the return type.
- Functions that return date and time typed data from components such as separately supplied year, month, and day. 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.

For details of all date and time functions, see Books Online 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. Ensure that you have completed the previous demonstration in this module. Alternatively, start the MSL-TMG1, 20761A-MIA-DC, and 20761A-MIA-SQL virtual machines, then log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. If SQL Server Management Studio is not already open, start it and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod06\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. In the **Available Databases** list, click **AdventureWorksLT**.
5. Select the code under the comment **Step 2**, and then click **Execute**.
6. Select the code under the comment **Step 3**, and then click **Execute**.
7. Select the code under the comment **Step 4**, and then click **Execute**.

8. Select the code under the comment **Step 5**, and then click **Execute**.
9. Select the code under the comment **Step 6**, and then click **Execute**.
10. Select the code under the comment **Step 7**, and then click **Execute**.
11. 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 data and time data.
- Write queries that use data and time functions.
- Write queries that return character data.
- Write queries that use character functions.

Estimated Time: 90 Minutes

Virtual machine: **20761A-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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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

1. 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.

   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 project file D:\Labfiles\Lab06\Starter\Project\Project.ssmssln and the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement against the Sales.Customers table and retrieve the contactname and city columns. Concatenate both columns so that the new column looks like this:

```
Allen, Michael (city: Berlin)
```

3. Execute the written statement and compare 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 project file D:\Labfiles\Lab06\Starter\Project\Project.ssmssln and the T-SQL script **81 - Lab Exercise 4.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the contactname column from the Sales.Customers table. Based on this column, add a calculated column named lastname, which should consist of all the characters before the comma.
3. Execute the written statement and compare 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.

▶ 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

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 7-13

Lab: Using DML to Modify Data 7-16

Module Review and Takeaways 7-19

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, UPDATE to make changes to column values to rows within tables, and DELETE to remove complete rows from tables. There is also the command TRUNCATE that you can use 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 a command named MERGE 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 how this impacts 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

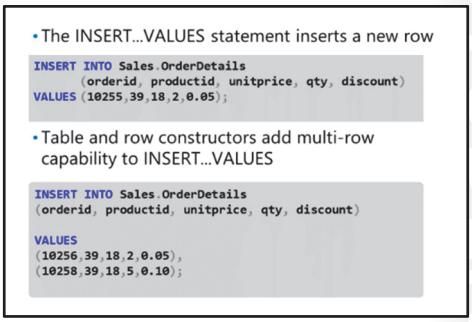
After completing this lesson, you will be able to:

- 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:



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)

```
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);
```

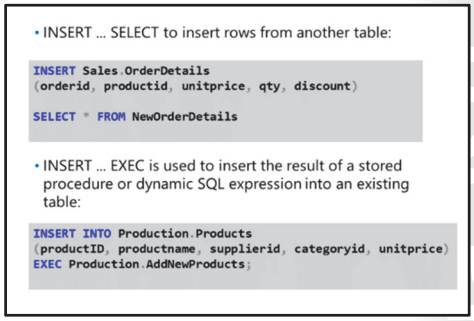
INSERT (Transact-SQL)

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 may 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.

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:

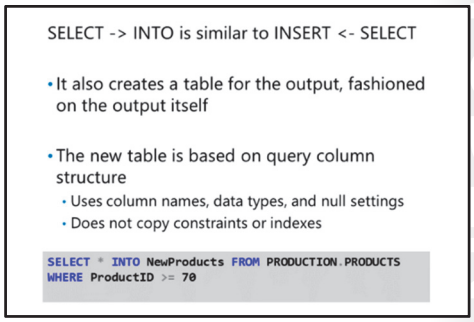
Inserting rows into a table from a stored procedure

```
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 column1

, column2

...

INTO NewTable FROM OldTable

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

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, as 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_SomeStorerdProcedure
	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 from that is produced by a stored Procedure.
- How to use the SELECT INTO keywords to insert data into a table.

Demonstration Steps

INSERT Data into a Table

1. Start the **20761AD-MIA-DC** and **20761A-MIA-SQL** virtual machines, log on to **20761A-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**,
2. Run **D:\Demofiles\Mod07\Setup.cmd** as an administrator. Click **Yes** when prompted.
3. In the Command Prompt window press **y**, and then press Enter.
4. When the script has finished, press Enter.
5. Open **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\Mod07\Demo** folder.
7. In Solution Explorer, expand **Queries**, and double-click **11 - Demonstration A.sql**.

8. Highlight the code **USE TSQL GO**, and click **Execute**.
9. First we 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 the procedure is not there.

10. Highlight the code below the comment **--Create a backup of the Products with a chosen ID**

```
DROP TABLE IF EXISTS NewProducts
GO
SELECT * INTO NewProducts
FROM PRODUCTION.PRODUCTS WHERE ProductID >= 70
```

    We are creating a new Table for NewProducts where the Product ID >= 70.

11. We 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 **Excute.**

```
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
```

12. Highlight the code below the **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 we were previously missing when we tried to run it at the beginning of the demo.

13. Now we need to populate the original products table with the data within the secondary table as if it was new rows that we are adding. 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
```

And click **Execute**, to transfer the rows and see that they have been transferred.

14. For the other table we 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
```

15. Having seen various ways to add data to a new or existing table, we 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
```

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.

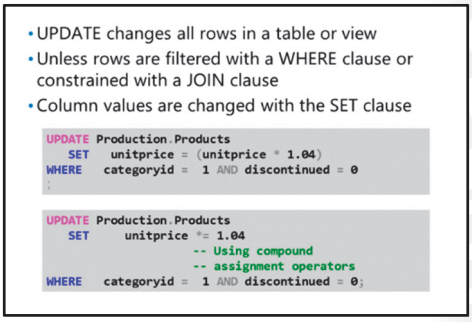
Lesson Objectives

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 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.



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:

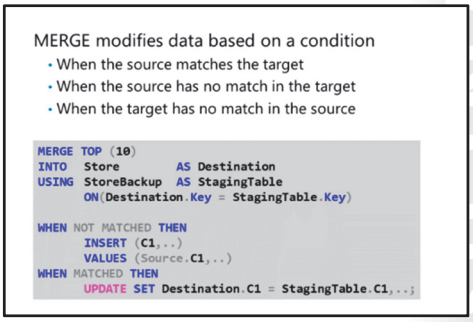
```
UPDATE Production.Products
    SET unitprice  *= 1.04
WHERE categoryID = 1
AND     discontinued = 0;
```

UPDATE (Transact-SQL)

http://aka.ms/sbikqm

Using MERGE to Modify Data

In database operations, there is a common need to perform an 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>);
```

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 **20761AD-MIA-DC** and **20761A1D-MIA-SQL** virtual machines, log on to **20761A-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**,
2. Run **D:\Demofiles\Mod07\Setup.cmd** as an administrator.
3. In the Command Prompt window press **y**, and then press **Enter**.
4. When the script has finished, press **Enter**.
5. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication,
6. Open the **Demo.ssmssln** solution in the D:\Demofiles\Mod07\Demo folder.
7. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
8. Highlight the code **USE AdventureWorks GO**, and click **Execute**.
9. Select the code under **USE AdventureWorks GO**, and then click **Execute**.
10. Select the code under the comment **Remove the copied rows from the store table**, and then click **Execute**.
11. Select the code under the comment **Show that they have been removed**, and then click **Execute**.
12. Select the code under the comment **Use the Merge statement to put them back**, and then click **Execute**.
13. 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**.
14. Select the code under the comment **Use the Merge statement to Change the names back**, and then click **Execute**.
15. 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**.
16. Select the code under the comment **Clean up the database**, and then click **Execute**.
17. Close SQL Server Management Studio without saving any files.

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

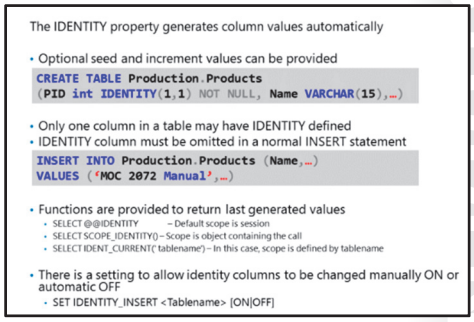
Lesson Objectives

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 as 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
, …
)
```

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 Books Online at:

SET IDENTITY_INSERT (Transact-SQL)

http://aka.ms/l4wavg

Once a value is assigned to a column by the IDENTITY property, the value may be retrieved like any other value in a column. Values generated by the IDENTITY property are unique within a table. However, without a constraint on the column (such as a PRIMARY KEY or UNIQUE constraint), uniqueness is not enforced after the value has been generated.

To return the most recently assigned value within the same session and scope, such as a stored procedure, use the SCOPE_IDENTITY() function. The legacy @@IDENTITY function will return the last value generated during a session, but it does not distinguish scope. You can use SCOPE_IDENTITY() for most purposes.

To reset the IDENTITY property by assigning a new seed, use the DBCC CHECKIDENT statement. See DBCC CHECKIDENT (Transact-SQL) in Books Online at:

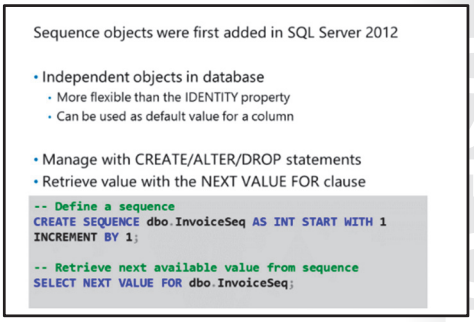
DBCC CHECKIDENT (Transact-SQL)

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 may be used to generate a sequence of values for a column within a table. However, the IDENTITY property is not suitable for coordinating values across multiple tables within a database. Database administrators and developers have needed to create tables of numbers manually to provide a pool of sequential values across tables.

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.

See CREATE SEQUENCE (Transact-SQL) in Books Online at:

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

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

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

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: **20761A-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

The exercises will be performed within the TempDB database so that none of the real data is affected. Two scripts are used to set up the environment for the lab—both are included in the project for the lab, along with a sample solution for each exercise. If you need to start again, open and execute the clean-up script, followed by the set-up script; you will be back to a clean environment and can try again.

1. Ensure that the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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 which 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 which 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

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
You are part way through the exercises and want to start again from the beginning. You run the set-up script within the solution and receive lots of error messages. This may occur if you have tried to execute the set-up script without running the clean-up script to remove any changes you may have made during the lab.	Run the clean-up script before running the set-up script.

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 8-14

Lesson 4: Using Functions to Work with NULL 8-18

Lab: Using Built-in Functions 8-22

Module Review and Takeaways 8-26

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.

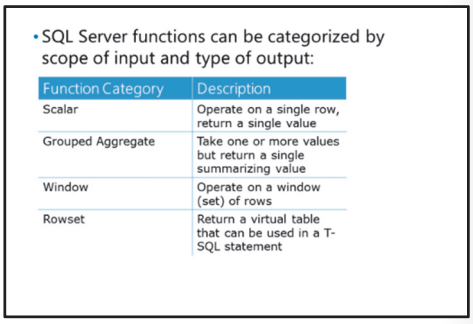
Lesson Objectives

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:



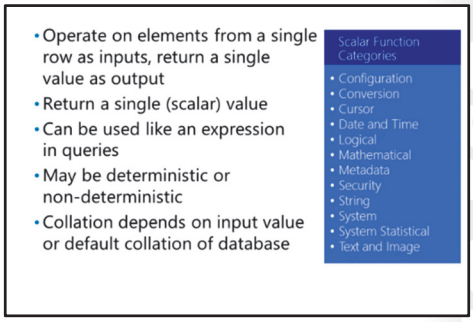
Function Category	Description
Scalar	Operate on a single row, return a single value
Grouped Aggregate	Take one or more input values, return a single summarizing value
Window	Operate on a window (set) of rows
Rowset	Return a virtual table that can be used in a T-SQL statement

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 may be used anywhere a single value (the result) could exist in its own right—from SELECT clauses to WHERE clause predicates.



Built-in scalar functions can be organized into many categories, such as string, conversion, logical, mathematical, and others. This lesson will look at a few common scalar functions.

Some considerations when using scalar functions include:

- **Determinism**: Will the function return the same value for the same input and database state each time? Many built-in functions are non-deterministic, and as such, their results cannot be indexed. This will have an impact on the query processor's ability to use an index when executing the query.
- **Collation**: When using functions that manipulate character data, which collation will be used? Some functions use the collation of the input value; others use the collation of the database if no input collation is supplied.

At the time of writing, Books Online 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
```

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
```

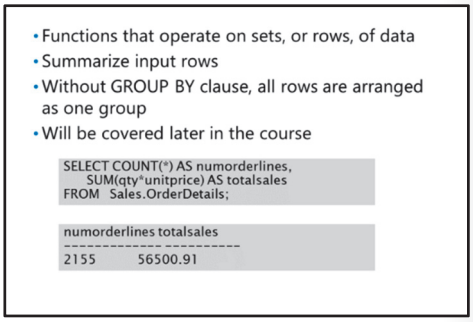
For additional information about scalar functions and categories, see Books Online at:

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:

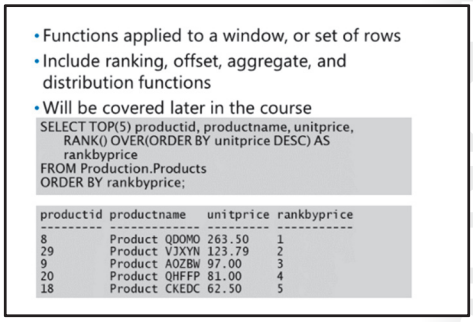
```
numorders   totalsales
----------- -----------
2155        56500.91
```

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:

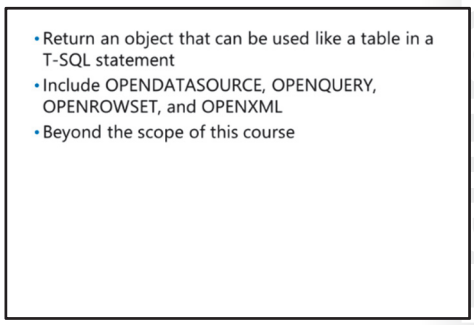
```
productid   productname   unitprice  rankbyprice
----------- ------------- ---------- -------------
38          Product QDOMO 263.50     1
29          Product VJXYN 123.79     2
9           Product AOZBW 97.00      3
20          Product QHFFO 81.00      4
18          Product CKEDC 62.50      5
```

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 Books Online at:

Rowset Functions (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402746

Demonstration: Writing Queries Using Built-in Functions

In this demonstration you will see how to:

- Use build-in scalar functions

Demonstration Steps

Use Built-in Scalar Functions

1. Ensure that the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run **D:\Demofiles\Mod08\Setup.cmd** as an administrator. Click **Yes** when prompted.
3. When prompted press **y**, and then press Enter.
4. When the script has finished 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\Mod08\Demo folder.
7. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
8. 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. Select the code under the comment **Step 4**, and then click **Execute**.
13. Keep SQL Server Management Studio open for the next demonstration.

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.

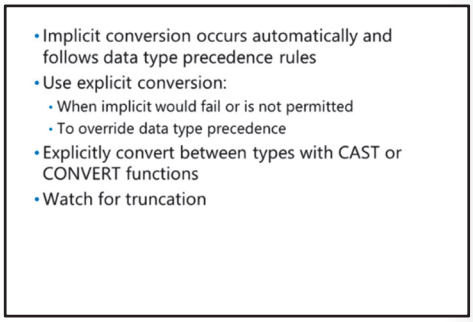
Lesson Objectives

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 may 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 non-character type to a character, the return value uses the collation of the database. The COLLATE option may be used with CAST or CONVERT to override this behavior.
- **Truncation**. When you convert data between character or binary types and different data types, data may be truncated, it might appear cut off, or an error could be thrown because the result is too short to display. 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 Books Online at:

CAST and CONVERT (Transact-SQL)

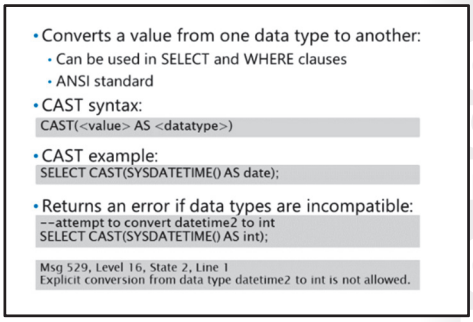
http://go.microsoft.com/fwlink/?LinkID=402747

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.
```

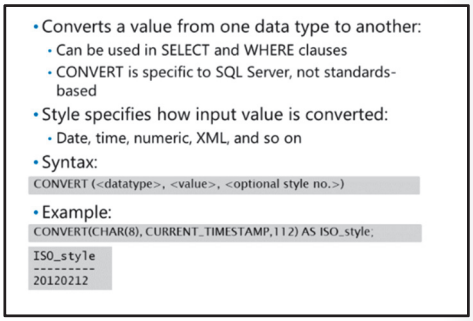
For more information about CATS, see Books Online at:

CAST and CONVERT (Transact SQL)

http://go.microsoft.com/fwlink/?LinkID=402747

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 yyyymmdd

The following example uses CONVERT to convert the current time from datetime to char(8):

CONVERT Example

```
SELECT CONVERT(CHAR(8), CURRENT_TIMESTAMP, 12) AS ISO_short, CONVERT(CHAR(8),
CURRENT_TIMESTAMP, 112) AS ISO_long;
```

The results:

```
ISO_short ISO_long
--------- --------
120212    20120212
```

For more information about CONVERT and its style options, see Books Online at:

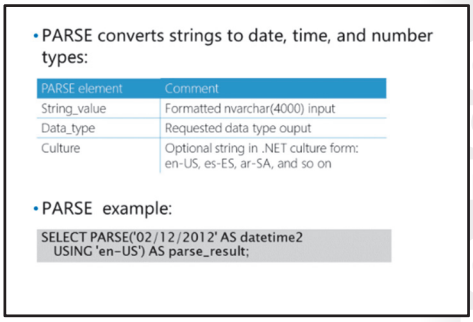
CAST and CONVERT (Transact-SQL)

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:



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 Books Online at:

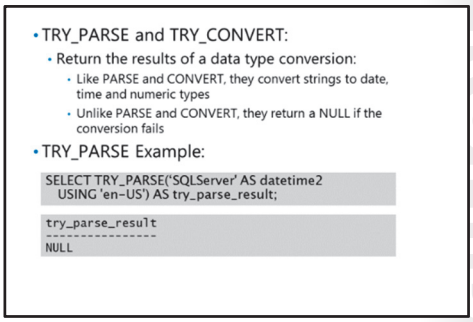
PARSE (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402732

Converting with TRY_PARSE and TRY_CONVERT

When using CONVERT or PARSE, an error may occur if the input value cannot be converted to the specified output type.

For example, if February 31, 2012 (an invalid date) is passed to CONVERT, a runtime error is raised:



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. If you have not completed the previous demonstration or need to start over, perform these two steps before proceeding to step 2; otherwise go to step 2:
   a. Start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod08\Setup.cmd as an administrator.
   b. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod08\Demo folder.
2. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
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 **THIS WILL FAIL at converting datetime2 to int**, and then click **Execute**.
8. Select the code under the comment **Step 5**,and then click **Execute**.
9. Select the code under the comment **Step 6**, and then click **Execute**.
10. Select the code under the comment **Step 7**, and then click **Execute**.
11. Select the code under the comment **Step 8**, and then click **Execute**.
12. Select the code under the comment **This will succeed**, and then click **Execute**.
13. 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

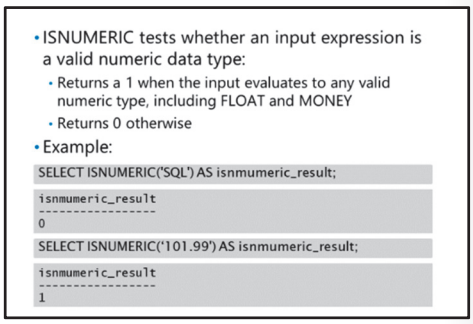
After completing this lesson, you will be able to:

- 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:



Writing Logical Tests with Functions

```
SELECT empid, lastname, postalcode
FROM HR.Employees
WHERE ISNUMERIC(postalcode)=1;
```

The results:

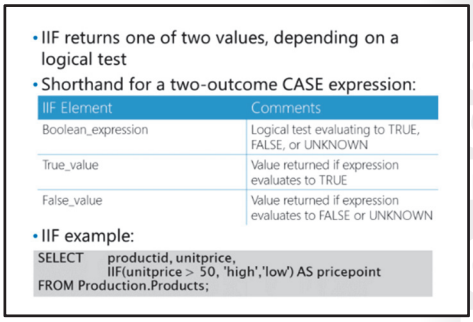
```
empid       lastname            postalcode
----------- ------------------- ----------
1           Davis               10003
2           Funk                10001
3           Lew                 10007
4           Peled               10009
5           Buck                10004
6           Suurs               10005
7           King                10002
8           Cameron             10006
9           Dolgopyatova        10008
```

Question: How might you use ISNUMERIC when testing data quality?

Performing Conditional Tests with IIF

IIF is a logical function in SQL Server. If you have used Visual Basic for Applications in Microsoft Excel®, used Microsoft Access®, or created expressions in SQL Server Reporting Services, you may have used IIF.

As in 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 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 Books Online at:

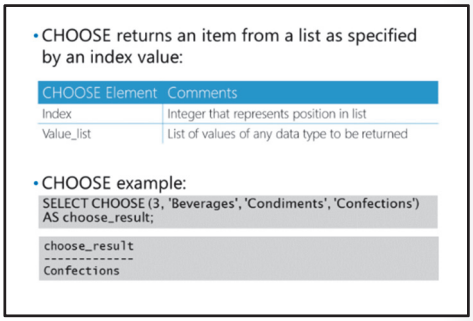
IIF (Transact-SQL)

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 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. If you have not completed the previous demonstration or need to start over, perform these two steps before proceeding to step 2; otherwise go to step 2:
   a. Start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod08\Setup.cmd as an administrator.
   b. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod08\Demo folder.
2. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
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. 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:

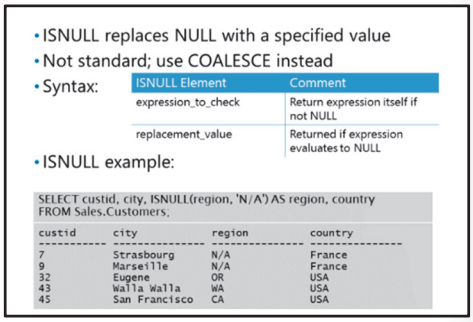
- Use ISNULL to replace NULLs.
- Use the COALESCE function to return non-NULL values.
- Use the NULLIF function to return NULL if values match.

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:



Converting NULL with ISNULL

```
SELECT custid, city, ISNULL(region, 'N/A') AS region, country
FROM Sales.Customers;
```

The result:

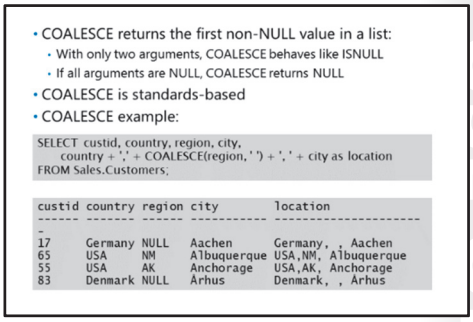
```
custid      city            region          country
----------- --------------- --------------- ---------------
40          Versailles      N/A             France
41          Toulouse        N/A             France
43          Walla Walla     WA              USA
45          San Francisco   CA              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 multi-part CASE expression using ISNULL.

If all arguments are NULL, COALESCE returns NULL.

The syntax is as follows:

COALESCE Syntax

```
SELECT COALESCE(<expression_1>[, ...<expression_n>];
```

The following example returns customers with regions where available, and adds a new column combining country, region and city, replacing NULL regions with a space:

COALESCE Example

```
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
```

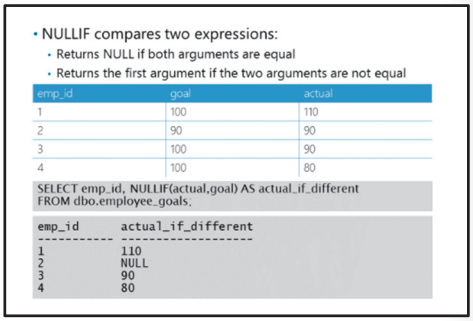
For more information on COALESCE and comparisons to ISNULL, see Books Online at:

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 Books Online at:

NULLIF (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402752

Demonstration: Using Functions to Work with NULL

In this demonstration, you will see how to:

- Use functions to work with NULL

Demonstration Steps

Use Functions to Work with NULL

1. If you have not completed the previous demonstration or need to start over, perform these two steps before proceeding to step 2; otherwise go to step 2:
   a. Start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod08\Setup.cmd as an administrator.

b. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod08\Demo folder.

2. In Solution Explorer, open the **41 – Demonstration D.sql** script file.
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 **First, set up sample data**, and then click **Execute**.
7. Select the code under the comment **Populate the sample data**, and then click **Execute**.
8. Select the code under the comment **Show the sample data**, and then click **Execute**.
9. Select the code under the comment **Use NULLIF to show which employees have actual values different from their goals**, and then click **Execute**.
10. Select the code under the comment **Step 5**, and then click **Execute**.

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: **20761A-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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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

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 9-10

Lesson 3: Filtering Groups with HAVING 9-16

Lab: Grouping and Aggregating Data 9-20

Module Review and Takeaways 9-26

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.

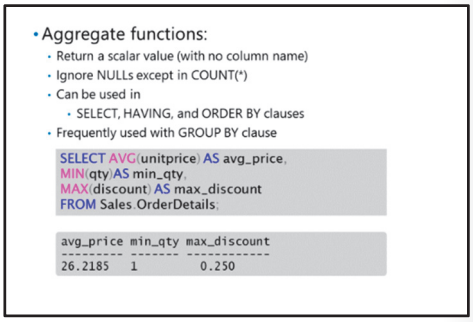
Lesson Objectives

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).

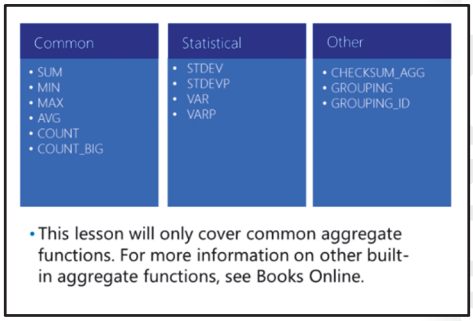
For more information on other built-in aggregate functions, see Books Online at:

Aggregate Functions (Transact-SQL)

http://aka.ms/wq6lku

Built-in Aggregate Functions

SQL Server provides many built-in aggregate functions. Commonly used functions include:



Function Name	Syntax	Description
SUM	SUM(<expression>)	Totals all the non-NULL numeric values in a column.
AVG	AVG(<expression>)	Averages all the non-NULL numeric values in a column (sum/count).
MIN	MIN(<expression>)	Returns the 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 Books Online at:

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. All columns, therefore, must be used as inputs to aggregate functions. Removing orderid from the previous example will prevent the error.

In addition to numeric data, such as the price and quantities in the previous example, aggregate expressions can also summarize date, time, and character data. The following examples show the use of aggregates with dates and characters:

This query returns first and last company by name, using MIN and MAX:

Aggregating Character Data

```
SELECT MIN(companyname) AS first_customer, MAX(companyname) AS last_customer
FROM Sales.Customers;
```

Returns:

first_customer last_customer

-------------- --------------

Customer AHPOP Customer ZRNDE

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:

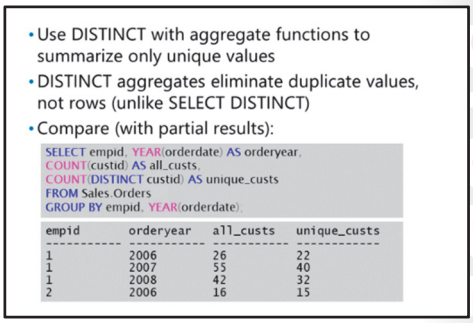
earliest latest

-------- -------

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:

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

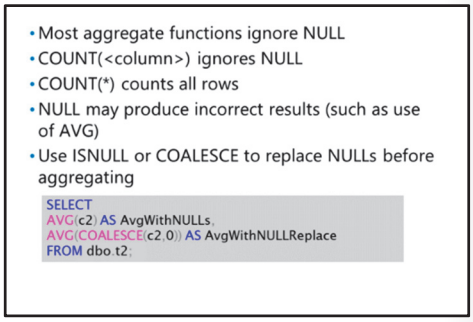
3 2008 38 30

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, given the following table named t1:

C1	C2
1	NULL
2	10
3	20
4	30
5	40
6	50

The following query illustrates the difference between how AVG handles NULL and how you might calculate an average with a SUM/COUNT(*) computed column:

Aggregating NULL Example

```
SELECT SUM(c2) AS sum_nonnulls,
      COUNT(*)AS count_all_rows,
      COUNT(c2)AS count_nonnulls,
      AVG(c2) AS [avg],
      (SUM(c2)/COUNT(*))AS arith_avg
FROM t1;
```

The result:

sum_nonnulls count_all_rows count_nonnulls avg arith_avg

------------ -------------- -------------- --- ---------

150 6 5 30 25

If you need to summarize all rows, whether NULL or not, consider replacing the NULLs with another value that can be used by your aggregate function.

The following example replaces NULLs with 0 before calculating an average. The table named t2 contains the following rows:

c1 c2

----------- -----------

1 1

2 10

3 1

4 NULL

5 1

6 10

7 1

8 NULL

9 1

10 10

11 1

12 10

Compare the effect on the arithmetic mean with NULLs-ignored verses replaced with 0.

Replace NULLs with Zeros Example.

```
SELECT AVG(c2) AS AvgWithNULLs, AVG(COALESCE(c2,0)) AS AvgWithNULLReplace
FROM dbo.t2;
```

This returns the following results, with a warning message:

```
AvgWithNULLs AvgWithNULLReplace
------------ ------------------
4            3
```

Warning: Null value is eliminated by an aggregate or other SET operation.

Note: This example cannot be executed against the sample database used in this course. 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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Run D:\Demofiles\Mod09\Setup.cmd as an administrator. Click **Yes** when prompted.
3. Start SQL Server Management Studio and connect to the MIA-SQL database instance using windows authentication.
4. On the **File** menu, point to **Open**, and then click **File**.
5. Browse to **D:\Demofiles\Mod09\Demo** folder, click **Demo.ssmssln**, and then click **Open**.
6. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
7. Expand the **Queries** folder, double-click the **11 – Demonstration A.sql** script file.
8. Select the code under the comment **Step 1: Using built-in Aggregate functions**, and then click **Execute**.
9. Select the code under the comment **Step 2: Using built-in Aggregate functions**, and then click **Execute**.
10. Select the code under the comment **Select and execute the following query to show This will succeed and return the AVG/MIN/MAX of all rows**, and then click **Execute**.
11. Select the first instance, under the comment **Select and execute the following query to show the use of aggregates with non-numeric data types**, and then click **Execute**.
12. Select the second instance, under the comment **Select and execute the following query to show the use of aggregates with non-numeric data types**, and then click **Execute**.
13. Select the code under the comment **Select and execute the following query to show the use of DISTINCT with aggregate functions**, and then click **Execute**.
14. Select the code under the comment **Select and execute the following query to show the impact of NULL on aggregate functions First, show the existence of NULLs in Sales.Orders**, and then click **Execute**.

15. Select the code under the comment **Then show that MIN, MAX and COUNT ignore NULL, COUNT(*) doesn't. Show the messages tab in the SSMS results pane for Warning: Null value is eliminated by an aggregate or other SET operation**, and then click **Execute**.
16. Select the code under the comment **Create an example table**, and then click **Execute**.
17. Select the code under the comment **Populate it**, and then click **Execute**.
18. Select the code under the comment **View the contents. Note the NULL**, and then click **Execute**.
19. Select the code under the comment **Execute this query to compare the behavior of AVG to an arithmetic average (SUM/COUNT)**, and then click **Execute**.
20. Select the code under the comment **Clean up the created table**, and then click **Execute**.
21. Select the code under the comment **Execute this query to demonstrate replacement of NULL before aggregating**, and then click **Execute**.
22. Select the code under the comment **Populate test table**, and then click **Execute**.
23. Select the code under the comment **Show table contents**, and then click **Execute**.
24. Select the code under the comment **Show standard AVG versus replacement of NULL with zero**, and then click **Execute**.
25. Select the code under the comment **clean up** 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.

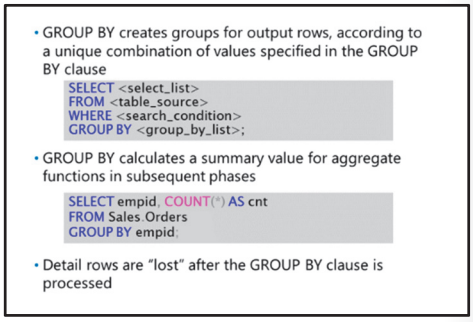
Lesson Objectives

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 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.

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 Books Online at:

GROUP BY (Transact-SQL)

http://aka.ms/ro266s

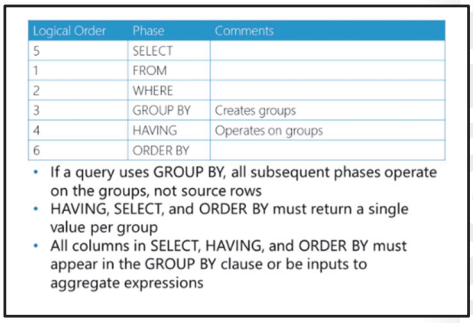
GROUP BY and the Logical Order of Operations

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.



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

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.

The following query is permitted because each column in the SELECT list is either a column in the GROUP BY clause or an aggregate function operating on each group:

GROUP BY Example.

```
SELECT empid, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
```

This returns:

empid count

----- -----

1 123

2 96

3 127

4 156

5 42

6 67

7 72

8 104

9 43

The following query will return an error since orderdate is not an input to GROUP BY, and its data has been "lost" following the FROM clause:

Missing GROUP BY Value.

```
SELECT empid, orderdate, COUNT(*) AS cnt
FROM Sales.Orders
GROUP BY empid;
```

This returns:

Msg 8120, Level 16, State 1, Line 1

Column 'Sales.Orders.orderdate' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

If you 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 date, using the T-SQL tools you have learned about so far. You will learn some approaches to solving the problem later in this course.

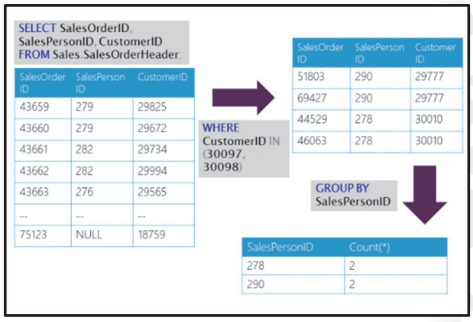
For more information about troubleshooting GROUP BY errors, see:

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

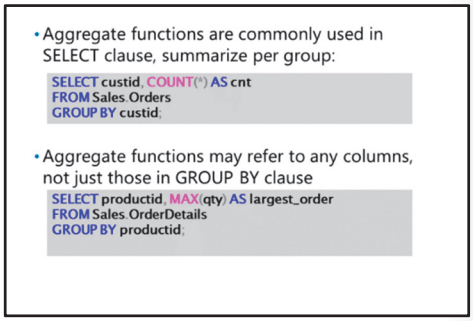
```
SELECT SalesOrderID, SalesPersonID,
CustomerID
FROM Sales.SalesOrderHeader;

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.

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. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod09\Setup.cmd as an administrator.

2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod09\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Select the code under the comment **Step 1: Using GROUP BY**, and then click **Execute**.
5. Select the code under the comment **Step 2: Using GROUP BY**, and then click **Execute**.
6. Select the code under the comment **Select this query and execute it to show customer orders per customer and per year**, and then click **Execute**.
7. Select the code under the comment **Step 3: Workflow of grouping**, and then click **Execute**.
8. Select the code under the comment **Step 4: Using Aggregates with GROUP BY**, and then click **Execute**.
9. Select the code under the comment **Show an aggregate on a column not in GROUP BY list**, and then click **Execute**.
10. 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.

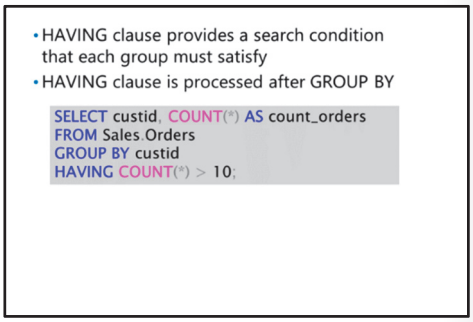
Lesson Objectives

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:



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:

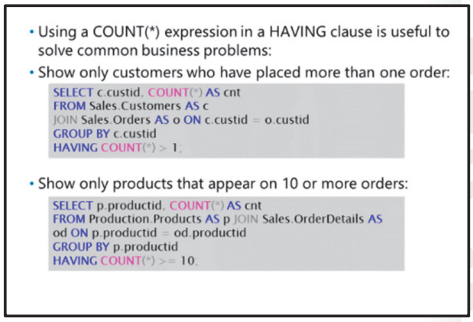
(28 row(s) affected)

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.

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. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod09\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod09\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. Select the code under the comment **Step 1: Filtering Groups with HAVING Change to Adventureworks database**, and then click **Execute**.
5. Select the code under the comment **Step 2: Using the HAVING clause**, and then click **Execute**.
6. Select the code under the comment **This query uses a HAVING clause to filter out customers with fewer than 10 orders**, and then click **Execute**.
7. Select the code under the comment **Review the logical order of operations**, and then click **Execute**.
8. Select the code under the comment **Select and execute the following queries to show difference between WHERE filter and HAVING filter**, and then click **Execute**.
9. Select the code under the comment **This query uses a HAVING clause to filter groups with an average quantity > 20**, and then click **Execute**.
10. Select the code under the comment **Select and execute the following query to show All customers and how many orders they have placed 89 rows**, and then click **Execute**.
11. Select the code under the comment **Use HAVING to filter only customers who have placed more than 20 orders**, and then click **Execute**.
12. Select the code under the comment **Select and execute the following query to show All products and in how many orders they appear**, and then click **Execute**.
13. Select the code under the comment **Use HAVING to filter only products which have been ordered less than 20 times, and then click Execute**.
14. 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: **20761A-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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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.

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.

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

Module Review and Takeaways

Review Question(s)

Question: What is the difference between the COUNT function and the COUNT_BIG function?

Question: Can a GROUP BY clause include more than one column?

Question: 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 **10-11**

Lab: Using Subqueries **10-14**

Module Review and Takeaways **10-19**

Module Overview

At this point in the course, you have learned many aspects of the T-SQL SELECT statement, but each query you have written has been a single, self-contained statement. Microsoft® SQL Server® 2016 also provides the ability to nest one query within another—in other words, to form subqueries. In a subquery, the results of the inner query (subquery) are returned to the outer query. This can provide a great deal of flexibility for your query logic. In this module, you will learn to write several types of subqueries.

Objectives

After completing this module, you will be able to:

- Describe the uses for queries that are nested within other queries.
- Write self-contained subqueries that return scalar or multi-valued results.
- Write correlated subqueries that return scalar or multi-valued results.
- Use the EXISTS predicate to efficiently check for the existence of rows in a subquery.

Lesson 1
Writing Self-Contained Subqueries

A subquery is a SELECT statement nested within another query. Being able to nest one query within another will enhance your ability to create effective queries in T-SQL. In this lesson, you will learn how to write self-contained queries, which are evaluated once, and provide their results to the outer query. You will learn how to write scalar subqueries, which return a single value, and multi-valued subqueries, which, as their name suggests, can return a list of values to the outer query.

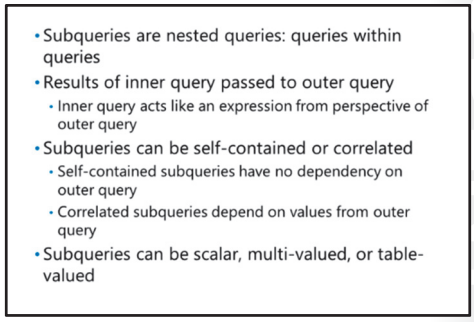
Lesson Objectives

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.

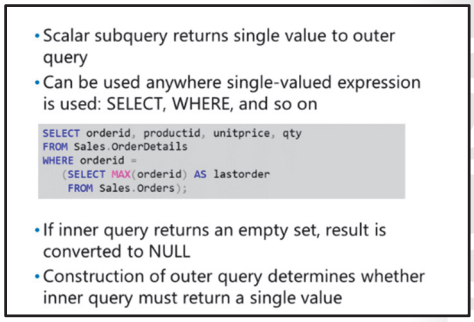
Note: You will learn about correlated subqueries later in this module.

For additional reading about subqueries, see Books Online at:

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
```

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 Books Online at:

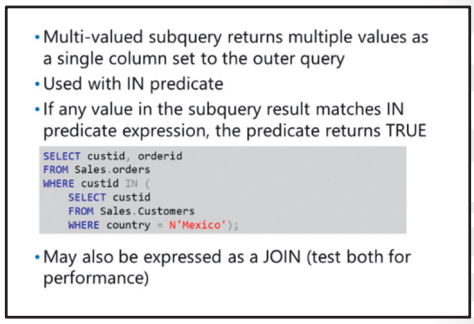
PRIMARY KEY Constraints

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

```
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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the **D:\Demofiles\Mod10** folder, right-click **Setup.cmd** and click **Run as administrator**.
3. Click **Yes** when prompted to confirm you want to run the command file, and wait for the script to finish.
4. When prompted 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**, click **File**, browse to the **D:\Demofiles\Mod10\Demo** folder, and then double-click **Demo.ssmssln**.
7. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.

8. Expand the **Queries** folder, and double-click the **11 – Demonstration A.sql** script file.
9. Select the code under the comment **Step 1: Open a new query window to the TSQL database**, and then click Execute.
10. Select the code under the comment **Step 2: Scalar subqueries**, and then click **Execute**.
11. Select the code under the comment **Select this query and execute it to find details in Sales.OrderDetails for most recent order**, and then click **Execute**.
12. Select the code under the comment **THIS WILL FAIL, since subquery returns more than 1 value**, and then click **Execute**.
13. Select the code under the comment **Step 3: Multi-valued subqueries**, and then click **Execute**.
14. Select the code under the comment **Same result expressed as a join**, 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, as well as rewrite some types of correlated subqueries as JOINs for performance or logical efficiency.

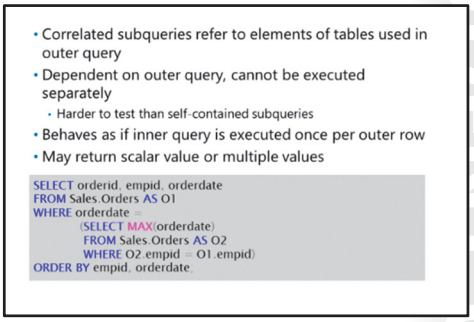
Lesson Objectives

After completing this lesson, you will be able to:

- Describe how correlated subqueries are processed.
- Write queries that use correlated subqueries in a SELECT statement.
- Rewrite some correlated subqueries as JOINs.

Working with Correlated Subqueries

Like self-contained subqueries, correlated subqueries are SELECT statements nested within an outer query. They may also be written as scalar or multi-valued subqueries. They are typically used to pass a value from the outer query to the inner query, to be used as a parameter there.



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;
```

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.

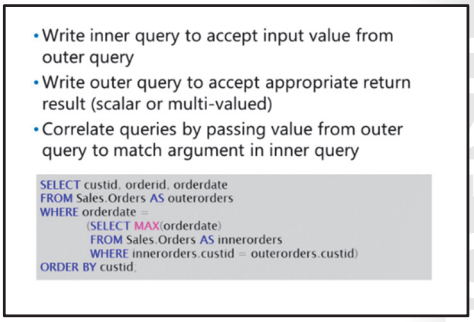
The query returns the following results. Note that some employees appear more than once, as they are associated with multiple orders on the latest orderdate:

```
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 may return multiple values, use an IN predicate. Plan to handle NULL results.
- Identify the column from the outer query that will be passed to the correlated subquery. Declare an alias for the table that is the source of the column in the outer query.
- Identify the column from the inner table that will be compared to the column from the outer table. Create an alias for the source table, as you did for the outer query.
- Write the inner query to retrieve values from its source, based on the input value from the outer query. For example, use the outer column in the WHERE clause of the inner query.

The correlation between the inner and outer queries occurs when the outer value is passed to the inner query for comparison. It's this correlation that gives the subquery its name.

For additional reading about correlated subqueries, see Books Online at:

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

1. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod10\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, and then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod10\Demo folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. Select the code under the comment **Step 1: Open a new query window to the TSQL database**, and then click **Execute**.
5. Select the code under the comment **Step 2: Correlated subqueries**, and then click **Execute**.

6. Select the code under the comment **Select and execute the following query to show the use of a correlated subquery that uses the empid from Sales.Orders to retrieve orders placed by an employee on the latest order date for each employee**, and then click **Execute**.
7. Select the code under the comment **Select and execute the following query to show the use of a correlated subquery that uses the custid from Sales.Custorders to retrieve orders placed by a customer with the highest quantity for each customer**, and then click **Execute**.
8. 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.

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.

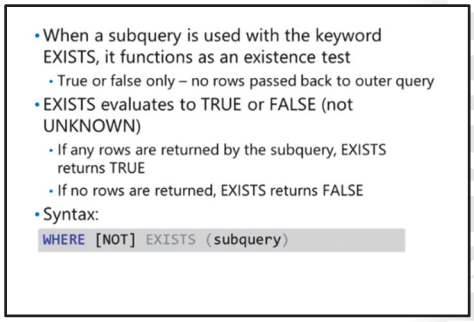
Lesson Objectives

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);
```

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

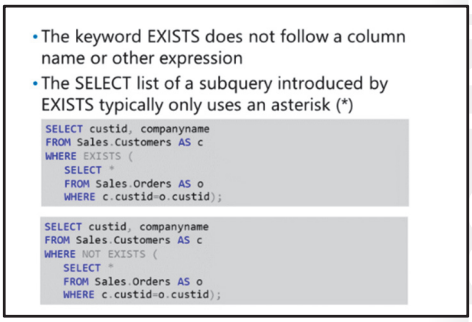
```
SELECT custid, companyname
FROM Sales.Customers AS c
WHERE NOT EXISTS (
      SELECT *
      FROM Sales.Orders AS o
      WHERE c.custid=o.custid);
```

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.



See *Subqueries with EXISTS* in Books Online at:

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. Ensure that you have completed the previous demonstration in this module. Alternatively, start the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**, and run D:\Demofiles\Mod10\Setup.cmd as an administrator.
2. If SQL Server Management Studio is not already open, start it and connect to the **MIA-SQL** database engine instance using Windows authentication, then open the **Demo.ssmssln** solution in the D:\Demofiles\Mod10\Demo folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. Select the code under the comment **Step 1: Open a new query window to the TSQL database**, and then click **Execute**.
5. Select the code under the comment **Step 2: Using EXISTS**, and then click **Execute**.
6. Select the code under the comment **Step 3: Using NOT EXISTS**, and then click **Execute**.
7. Select the code under the comment **Step 4: Compare COUNT(*)>0 to EXISTS**, and then click **Execute**.
8. Select the code under the comment **Use EXISTS**, and then click **Execute**.
9. 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 multi-result set subqueries.
- Write queries that use correlated subqueries and the EXISTS predicate.

Estimated Time: 60 minutes

Virtual machine: **20761A-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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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 Multi-Result Subqueries

Scenario

The marketing department would like to prepare materials for different groups of products and customers, based on historic sales information. You have to prepare different SELECT statements that use a subquery in the WHERE clause.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve Specific Products
2. Write a SELECT Statement to Retrieve Those Customers Without Orders
3. Add a Row and Rerun the Query That Retrieves Those Customers Without Orders

▶ Task 1: Write a SELECT Statement to Retrieve Specific Products

1. Open the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement to retrieve the **productid** and **productname** columns from the **Production.Products** table. Filter the results to include only products that were sold in high quantities (more than 100) for a specific order line.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab10\Solution\62 - Lab Exercise 2 -Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Retrieve Those Customers Without Orders

1. Write a SELECT statement to retrieve the **custid** and **contactname** columns from the **Sales.Customers** table. Filter the results to include only those customers who do not have any placed orders.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\63 - Lab Exercise 2 - Task 2 Result.txt. Remember the number of rows in the results.

▶ Task 3: Add a Row and Rerun the Query That Retrieves Those Customers Without Orders

1. The IT department has written a T-SQL statement that inserts an additional row in the **Sales.Orders** table. This row has a NULL in the **custid** column:

```
INSERT INTO Sales.Orders (
custid, empid, orderdate, requireddate, shippeddate, shipperid, freight,
shipname, shipaddress, shipcity, shipregion, shippostalcode, shipcountry)
VALUES
(NULL, 1, '20111231', '20111231', '20111231', 1, 0,
'ShipOne', 'ShipAddress', 'ShipCity', 'RA', '1000', 'USA')
```

2. Execute this query exactly as written inside a query window.
3. Copy the T-SQL statement you wrote in task 2 and execute it.
4. Observe the result. How many rows are in the result? Why?
5. Modify the T-SQL statement to retrieve the same number of rows as in task 2. (Hint: You have to remove the rows with an unknown value in the **custid** column.)
6. Execute the modified statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

Results: After this exercise, you should know how to use multi-result subqueries in T-SQL statements.

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

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 Set Operators

Contents:

Module Overview **11-1**

Lesson 1: Writing Queries with the UNION Operator **11-2**

Lesson 2: Using EXCEPT and INTERSECT **11-6**

Lesson 3: Using APPLY **11-10**

Lab: Using Set Operators **11-17**

Module Review and Takeaways **11-22**

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.

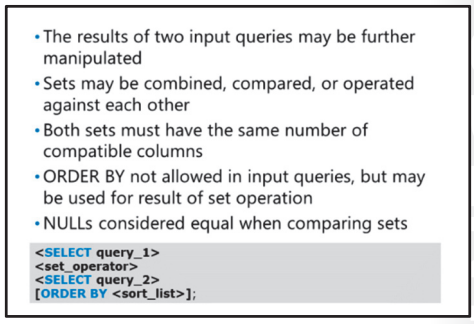
Lesson Objectives

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:

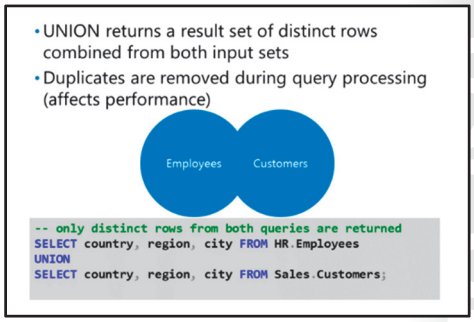


- Each input set is the result of a query, which may include any SELECT statement components you have already learned about, except an ORDER BY clause.
- The input sets must have the same number of columns and the columns must have compatible data types. The column data types, if not initially compatible, 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 six 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.

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.

See *UNION (Transact-SQL)* in Books Online at:

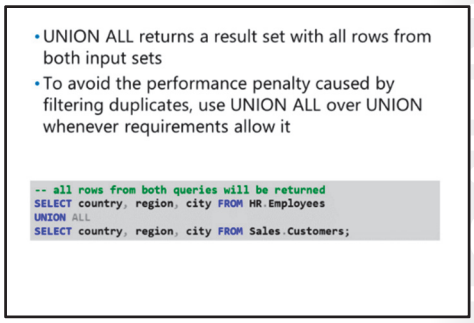
UNION (Transact-SQL)

http://aka.ms/omv6m7

Using the UNION ALL Operator

The UNION ALL operator works in a very 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 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, 20761A-MIA-DC, and 20761A-MIA-SQL virtual machines are running, and then log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. Start SQL Server Management Studio and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication.
3. If the **Microsoft SQL Server Management Studio** dialog box appears, click **OK**.
4. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod11\Demo** folder.
5. If the Solution Explorer pane is not visible, on the **View** menu, click **Solution Explorer**.
6. Expand **Queries**, and double-click the **11 – Demonstration A.sql** script file.
7. In the **Available Databases** list, click **AdventureWorksLT**.
8. Select the code under the comment **Step 2**, and then click **Execute**.
9. Select the code under the comment **Step 3**, and then click **Execute**.
10. 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
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.	

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 may be useful to your queries. You will learn how to use EXCEPT and INTERSECT in this lesson.

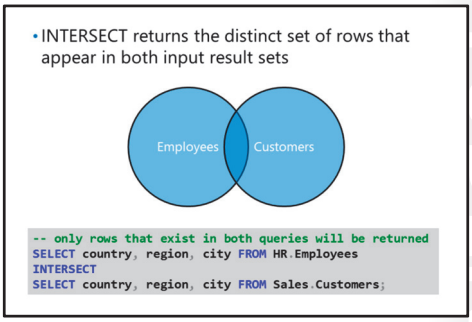
Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that use the EXCEPT operator to return only rows in one set but not another.
- Write queries that use the INTERSECT operator to return only rows that are present in both sets.

Using the INTERSECT Operator

The T-SQL INTERSECT operator, added in SQL Server 2005, returns only distinct rows that appear in both input sets.



Note: While UNION supports both 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   Sao Paulo
 (5 row(s) affected)
```

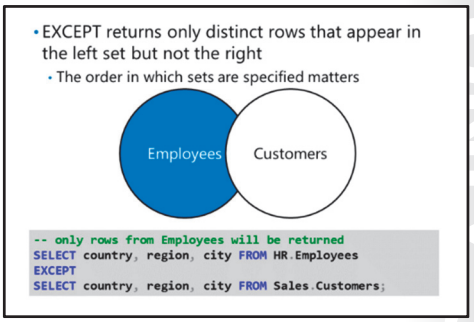
For more information, see *EXCEPT and INTERSECT (Transact-SQL)* in Books Online at:

EXCEPT and INTERSECT (Transact-SQL)

http://aka.ms/uo4qu9

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. Ensure that you have completed the previous demonstration in this module. Alternatively, start the MSL-TMG1, 20761A-MIA-DC, and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. If SQL Server Management Studio is not already open, start it and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication, and then open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod11\Demo** folder.
3. In Solution Explorer, open the **21 – Demonstration B.sql** script file.
4. In the **Available Databases** list, click **AdventureWorksLT**.
5. Select the code under the comment **Step 2**, and then click **Execute**.
6. Select the code under the comment **Step 3**, and then click **Execute**.
7. Select the code under the comment **Step 4**, and then click **Execute**.
8. Select the code under the comment **Step 5**, and then click **Execute**.
9. Select the code under the comment **Step 6**, and then click **Execute**.
10. 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.

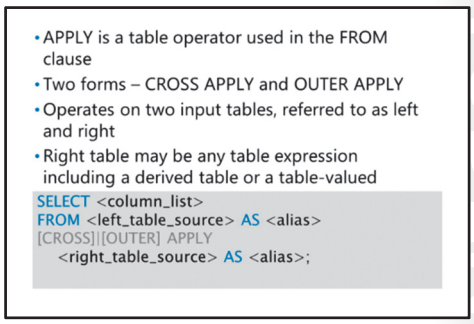
Lesson Objectives

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 table-valued function (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 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>;
```

See Using APPLY in the "Remarks" section of *FROM (Transact-SQL)* in Books Online at:

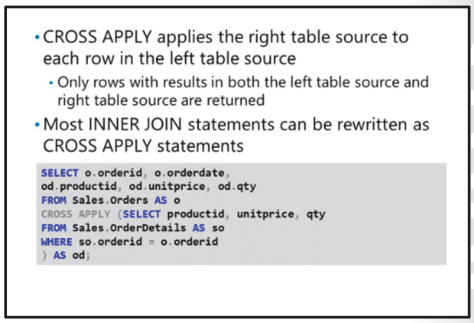
FROM (Transact-SQL)

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.



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

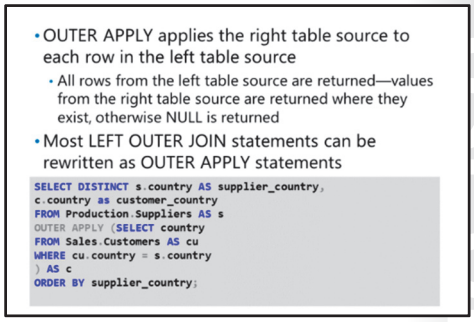
```
SELECT o.orderid, o.orderdate,
od.productid, od.unitprice, od.qty
FROM Sales.Orders AS o
CROSS APPLY ( SELECT productid, unitprice, qty
              FROM Sales.OrderDetails AS so
              WHERE so.orderid = o.orderid
                ) AS od;
```

 **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.

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

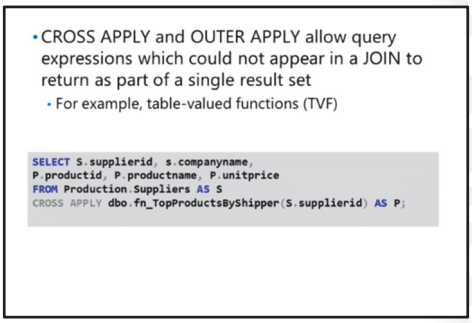
```
SELECT DISTINCT s.country AS supplier_country, c.country as customer_country
FROM Production.Suppliers AS s
OUTER APPLY ( SELECT country
                           FROM Sales.Customers AS cu
                           WHERE cu.country = s.country
                     ) AS c
ORDER BY supplier_country;
```

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 are queries which 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.

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. Ensure that you have completed the previous demonstration in this module. Alternatively, start the MSL-TMG1, 20761A-MIA-DC, and 20761A-MIA-SQL virtual machines, log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. If SQL Server Management Studio is not already open, start it and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication, and then open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod11\Demo** folder.
3. In Solution Explorer, open the **31 – Demonstration C.sql** script file.
4. In the **Available Databases** list, click **AdventureWorksLT**.
5. Select the code under the comment **Step 2**, and then click **Execute**.
6. Select the code under the comment **Step 3**, and then click **Execute**.
7. Select the code under the comment **Step 4**, and then click **Execute**.
8. Select the code under the comment **Step 5**, and then click **Execute**.
9. Select the code under the comment **Test with CROSS APPLY**, and then click **Execute**.
10. Select the code under the comment **Step 6**, and then click **Execute**.
11. Select the code under the comment **Step 7**, and then click **Execute**.
12. Select the code under the comment **Use OUTER APPLY to include customers with no orders**, and then click **Execute**.
13. 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: **20761A-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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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 Specific Products

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** 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\Lab11\ 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\Lab11\ 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\Lab11\ 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\Lab11\ 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\Lab11\ 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\Lab11\ 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\Lab11\ 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\Lab11\ 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\Lab11\ 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

▶ 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\Lab11\ 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\Lab11\ 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\Lab11\ 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\Lab11\ 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\Lab11\ 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.

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.

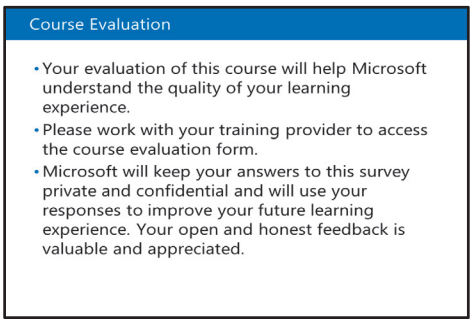
Review Question(s)

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 was empty?

Question: Which form of the APPLY operator can be used to rewrite LEFT OUTER JOIN queries?

Course Evaluation



Your evaluation of this course will help Microsoft understand the quality of your learning experience.

Please work with your training provider to access the course evaluation form.

Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.

Module 1: 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 **20761A-MIA-DC** and **20761A-MIA-SQL** virtual machines are running.
2. Log on to **20761A-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
3. Start SQL Server Management Studio.
4. In the **Connect to Server** window, 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. Open the **Object Explorer** window by selecting **Object Explorer** on the **View** menu (or press **F8**).
8. Open the **Solution Explorer** window by selecting **Solution Explorer** on the **View** menu (or press **Ctrl+Alt+L**).

▶ Task 2: Configure the Editor Settings

1. With SQL Server Management Studio open, on the **Tools** menu, click **Options**.
2. Expand the **Environment** option and click **Fonts and Colors**. In the **Show settings for** dialog box, select **Text Editor**. Set the font size to **14** in the **Size** box.
3. In the left pane, expand the **Text Editor** option, then expand the **Transact-SQL** option, and select **IntelliSense** to display the **Transact-SQL IntelliSense Settings.** Clear the **Enable IntelliSense** check box.
4. In the left pane, select **Tabs** (If you have closed the tree, navigate to Text Editor, then Transact-SQL). Change the **Tab size** to **6**.
5. In the left pane, expand **Query Results**, then expand **SQL Server**, and select **Results to Grid**. Enable the option **Include column headers when copying or saving the results**.
6. Apply the changes by clicking the **OK** button.

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

Exercise 2: Creating and Organizing T-SQL Scripts

▶ Task 1: Create a Project

1. Select the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project** window, select **SSMSEmptySqlProject.**
3. Type **MyFirstProject** in the **Name** text box and **D:\Labfiles\Lab01\Starter** in the **Location** text box. Click the **OK** button to create the new project.

4. In the Solution Explorer window, right-click the **Queries** folder under MyFirstProject and click **New Query**.
5. In the **Connect to Database Engine** dialog box, click **Cancel**.
6. Right-click the query file **SQLQuery1.sql** under the **Queries** folder, click **Rename** and type **MyFirstQueryFile.sql** as the new name for the file.
7. On the **File** menu, click **Save All**.

▶ Task 2: Add an Additional Query File

1. In the **Solution Explorer** window, right-click the **Queries** folder under **MyFirstProject**. click **New Query**.
2. In the **Connect to Database Engine** dialog box, click **Cancel**.
3. In the **Queries** folder, right-click the query file **SQLQuery1.sql,** click **Rename** and type **MySecondQueryFile.sql** as the new name for the file.
4. Click **File Explorer** on the taskbar.
5. In Windows File Explorer, navigate to the folder **D:\Labfiles\Lab01\Starter\MyFirstProject\MyFirstProject** to see where the files have been created.
6. In SQL Server Management Studio, in the **Solution Explorer** window, right-click on the query file **MySecondQueryFile.sql,** and select **Remove**. When the confirmation dialog appears, click **Remove**.
7. In Windows Explorer, press **F5** to refresh the Windows Explorer window to see that the file **MySecondQueryFile.sql** is still there.
8. In SQL Server Management Studio Solution, right-click the query file **MyFirstQueryFile.sql** and select **Remove**. When the confirmation dialog appears, click **Delete**.
9. In Windows Explorer, press F5 to refresh the Windows Explorer window. Notice that the file **MyFirstQueryFile.sql** has been deleted.

▶ Task 3: Reopen the Created Project

1. 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 again, and in the **Connect to Server** window, click **Cancel**.
4. On the **File** menu, point to **Open** and click **Project/Solution**. In the **Open Project** window, select the project **D:\Labfiles\Lab01\Starter\MyFirstProject\MyFirstProject.ssmssln** and click **Open**.
5. In Windows Explorer, navigate to the folder **D:\Labfiles\Lab01\Starter\MyFirstProject\MyFirstProject**.
6. Drag the file **MySecondQueryFile.sql** to the **Queries** folder in **Solution Explorer**.
7. On the **File** menu, select **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 virtual machine, press the **Windows** key, type **manage help settings**, and then click **Manage Help Settings**.
2. In the **Help Library Manager** window, select **Choose online or local help**.
3. Under **Set your preferred help experience**, click **I want to use online help**, and then click the **OK** button to confirm. If **I want to use online help**, is already selected then press **Cancel**.
4. Click the **Exit** button to close the **Help Library Manager** window.

▶ Task 2: Use SQL Server 2016 Technical Documentation

1. On the virtual machine, press the **Windows** key, type **sql server documentation,** and then click **SQL Server Documentation**.
2. If the **Online Help Consent** dialog box appears, click **Yes** to continue.
3. In the left pane, or the top menu of the **SQL Server 2016 Technical Documentation** website, expand **SQL Server 2016 Technical Documentation**, and then expand **Database Engine**.
4. Click **What's New in Database Engine**.
5. Browse the help article.
6. Close Internet Explorer.

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

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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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.
1. 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** window, 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** window, browse to the **D:\Labfiles\Lab02\Starter\Project** folder, and then double-click **Project.ssmssln**.
5. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, on the **View** menu, click **Solution Explorer**.
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** (or press F5). It is very important to understand that you can highlight a specific part of the code inside the T-SQL script, and execute 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.
3. Close Microsoft SQL Server Management Studio. If prompted to save the files, click **No**.

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** window, 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** window, 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. When the query window opens, 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 column country. Only rows for which the logical expression evaluates to **TRUE** are returned by the **WHERE** phase to the subsequent logical query processing phase.

▶ Task 3: 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.

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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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** window, in the **Server name** text 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, choose **Yes**.
5. Under **User Databases**, select the **TSQL** database, and then click **OK**.
6. On the **Login** tab, in the **Authentication** list, click **Windows Authentication**, and click **Connect**.
7. In Object Explorer, expand the server **MIA-SQL**, expand **Databases**, expand the database **TSQL**, and 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 click **Project/Solution**.
2. In the **Open Project** window, open the project **D:\Labfiles\Lab03\Starter\Project\Project.ssmssln**.
3. In Solution Explorer, expand **Queries**, and double-click **Lab Exercise 1.sql**. (If Solution Explorer is not visible, on the **View** menu, click **Solution Explorer**.
4. When the query window opens, highlight the statement **USE TSQL;**, and 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 in step 5 and click **Execute**.

7. In the query pane, type the following code after the first query:

```
SELECT *
FROM
```

8. In Object Explorer, select the **Sales.Customers** table under **MIA-SQL**, **TSQL**, **Tables**. Using the mouse, drag the selected table into the query pane, after the FROM clause. Add a semicolon to the end of the SELECT statement.
9. 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 the **Sales.Customers** table.
2. Expand **Columns** and observe all the columns in the **Sales.Customers** table.
3. In the query pane, after the task 3 description, type the following query:

```
SELECT contactname, address, postalcode, city, country
FROM Sales.Customers;
```

4. Highlight the written query and click **Execute**.
5. Observe the result. How many rows are affected by the last query? There are multiple ways to answer this question using SQL Server Management Studio. One way is to select the previous query and click **Execute**. The total number of rows affected by the executed query is written in the **Results** pane under the **Messages** tab:

```
(91 row(s) affected)
```

Another way is to look at the status bar displayed below the **Results** pane. On the left side of the status bar, there is a message stating: "Query executed successfully." On the right side, the total number of rows affected by the current query is displayed (91 rows).

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

Exercise 2: Eliminating Duplicates Using DISTINCT

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

1. In Solution Explorer, double-click **Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;**, and 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. On the **Edit** menu, click **Copy**.
2. In the query window, click the line after the task 2 description. On the **Edit** menu, click **Paste**. You have now copied the previous query to the same query window after the task 2 description.
3. Modify the query by typing **DISTINCT** after the SELECT clause. Your query should look like this:

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

4. Highlight the written query and click **Execute**.
5. Observe the result and answer these questions:

How many rows did the query in task 1 return?

*To answer this question, you can highlight the query written under the task 1 description, click **Execute**, and read the **Results** pane. (If you forgot how to access this pane, look at task 4 in exercise 1.) The number of rows affected by the query is 91.*

How many rows did the query in Task 2 return?

*To answer this question, you can highlight the query written under the task 2 description, click **Execute**, and read the **Results** pane. The number of rows affected by the query is 21. This means that there are 21 distinct values for the **country** column in the **Sales.Customers** table.*

Under which circumstances do the following queries against the **Sales.Customers** table return the same result?

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

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. When the query window opens, 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. When the query window opens, highlight the statement **USE TSQL;**, and 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. On the **Edit** menu, click **Copy**.
2. In the query window, click the line after the task 3 description. On the **Edit** menu, click **Paste**. You have now copied the previous query to the same query window after the task 3 description.
3. Add a new column using an additional CASE expression. Your query should look like this:

```
SELECT    p.categoryid, p.productname,
  CASE
         WHEN p.categoryid = 1 THEN 'Beverages'
         WHEN p.categoryid = 2 THEN 'Condiments'
WHEN p.categoryid = 3 THEN 'Confections'
         WHEN p.categoryid = 4 THEN 'Dairy Products'
         WHEN p.categoryid = 5 THEN 'Grains/Cereals'
         WHEN p.categoryid = 6 THEN 'Meat/Poultry'
         WHEN p.categoryid = 7 THEN 'Produce'
         WHEN p.categoryid = 8 THEN 'Seafood'
         ELSE 'Other'
  END AS categoryname,
  CASE
         WHEN p.categoryid IN (1, 7, 8) THEN 'Campaign Products'
         ELSE 'Non-Campaign Products'
```

```
   END AS iscampaign
FROM Production.Products AS p;
```

4. Highlight the written query and click **Execute**.
5. In the result, observe that the first CASE expression uses the simple form, whereas the second uses the searched form.

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

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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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**, when prompted, type **y**, 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** window, in the **Server name** box, type **MIA-SQL**, and 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. If Solution Explorer is not visible, on the **View** menu, click **Solution Explorer**.
6. In Solution Explorer, expand Queries, and then double-click the **51 - Lab Exercise 1.sql** query.
7. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
8. 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;
```

9. Highlight the written query and click **Execute**.
10. 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

1. In Solution Explorer, double-click the **61 - Lab Exercise 2.sql** query.
2. In the query window, highlight the statement **USE TSQL;**, and then click **Execute**.
3. Under the Task 1 description, highlight the written query, and click **Execute**.
4. Observe the error message:

   Ambiguous column name 'custid'.

5. 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

1. Highlight the previous query. On the **Edit** menu, click **Copy**.
2. In the query window, click the line after the Task 2 description. On the **Edit** menu, click **Paste**.
3. 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;
```

4. Highlight the modified query and click **Execute**.

▶ Task 3: Change the Table Aliases

1. Highlight the previous query. On the **Edit** menu, click **Copy**.
2. In the query window, click the line after the Task 3 description. On the **Edit** menu, click **Paste**.
3. Modify the **T-SQL** statement to use table aliases. Your query should look like this:

```
SELECT
c.custid, c.contactname, o.orderid
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid;
```

4. Highlight the written query and click **Execute**.
5. Compare the results with the Task 2 results.

6. Modify the **T-SQL** statement to include a full source table name as the column prefix. Your query should now look like this:

```
SELECT
Customers.custid, Customers.contactname, Orders.orderid
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid;
```

7. Highlight the written query and click **Execute**.
8. Observe the error messages:

   Msg 4104, Level 16, State 1, Line 57

   The multi-part identifier "Customers.custid" could not be bound.

   Msg 4104, Level 16, State 1, Line 57

   The multi-part identifier "Customer.contactname" could not be bound.

   Msg 4104, Level 16, State 1, Line 57

   The multi-part identifier "Orders.orderid" could not be bound.

   You received these error messages as, because you are using a different table alias, the full source table name you are referencing as a column prefix no longer exists. Remember that the SELECT clause is evaluated after the FROM clause, so you must use the table aliases when specifying columns in the SELECT clause.

9. Modify the SELECT statement so that it uses the correct table aliases. Your query should look like this:

```
SELECT
c.custid, c.contactname, o.orderid
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid;
```

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. When the query window opens, 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. On the **Edit** menu, click **Copy**.
2. In the query window, after the Task 2 description, click the line. 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

1. 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.

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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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. In the **Command Prompt** window, 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 click **Project/Solution**.
3. In the **Open Project** window, open the project: **D:\Labfiles\Lab05\Starter\Project\Project.ssmssln**.
4. If Solution Explorer is not visible, on the **View** menu, select **Solution Explorer** or press Ctrl+Alt+L on the keyboard.
5. In Solution Explorer, expand **Queries**, and double-click **51 - Lab Exercise 1.sql**.
6. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
7. 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';
```

8. 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. On the toolbar, click **Edit** and then **Copy**.
3. In the query window, click the line after the task 4b description. On the toolbar, click **Edit** and then **Paste**. You have now copied the previous query to the same query window after the task 4b description.
4. Modify the query so that it looks like this:

```
SELECT
c.custid, c.companyname, o.orderid
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON c.custid = o.custid
WHERE
c.city = N'Paris';
```

5. Highlight the modified query and click **Execute**.
6. Observe the result. Is it the same as that of the first SQL statement?

   The result is not the same. When you specify the predicate in the ON clause, the left outer join preserves all the rows from the left table (**Sales.Customers**) and adds only the matching rows from the right table (**Sales.Orders**), based on the predicate in the ON clause. This means that all the customers will show up in the output, but only the ones from Paris will have matching orders. When you specify the predicate in the WHERE clause, the query will filter only the Paris customers. So be aware that, when you use an outer join, the result of a query 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 non-preserved side to match to those from the preserved side. The WHERE predicate is a filtering predicate—if a row from either side doesn't satisfy the WHERE predicate, the row is filtered out.

▶ Task 6: Write a SELECT Statement to Retrieve 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 non-preserved 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 the query **61 - Lab Exercise 2.sql**. (If the solution is not already open, on the **File** menu, click **Open** and click **Project/Solution**. Then in the **Open Project** window, open the project **D:\Labfiles\Lab05\Starter\Project\Project.ssmssln**.)
2. When the query window opens, highlight the statement **USE TSQL**; and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
c.custid, c.contactname, o.orderid, o.orderdate
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid
WHERE
o.orderdate >= '20080401'
ORDER BY
o.orderdate DESC, c.custid ASC;
```

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. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 3a description. On the toolbar, click **Edit** and then **Paste**. You have now copied the previous query to the same query window after the task 3a description.
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. On the toolbar, click **Edit** and then **Copy**.
6. In the query window, click the line after the task 3b description. On the toolbar, click **Edit** and then **Paste**. You have now copied the previous query to the same query window after the task 3b description.
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 the query **71 - Lab Exercise 3.sql**. (If the solution is not already open, on the **File** menu, click **Open** and click **Project/Solution**. Then in the **Open Project** window, open the project **D:\Labfiles\Lab05\Starter\Project\Project.ssmssln**.)
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT TOP (20)
orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

4. Highlight the 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 the query **81 - Lab Exercise 4.sql**. (If the solution is not already open, on the **File** menu, click **Open** and click **Project/Solution**. Then in the **Open Project** window, open the project **D:\Labfiles\Lab05\Starter\Project\Project.ssmssln**.)
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
custid, orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate, orderid
OFFSET 0 ROWS FETCH FIRST 20 ROWS ONLY;
```

4. Highlight the 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 **Execeute**.

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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the **D:\Labfiles\Lab06\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish, 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 click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab06\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and double-click **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, on the **View** menu, click **Solution Explorer**.
5. When the query window opens, highlight the statement **USE TSQL;** and 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**.

▶ 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. Do not worry if you do not recognize the type conversion functions, as they will be covered in the next module.

3. Highlight the written queries and click **Execute**.
4. Observe the result and answer these questions:

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

  There are two main differences. First, the SYSDATETIME function provides a more precise time element compared to the CURRENT_TIMESTAMP function. Second, the SYSDATETIME function returns the data type **datetime2**(7), whereas the CURRENT_TIMESTAMP returns the data type **datetime**.

- What is a language-neutral format for the data type date?

  You can use the 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 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, 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 which 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 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, 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
```

5. 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 the query **81 - Lab Exercise 4.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, 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

The exercises will be performed within the TempDB database so that none of the real data is affected. Two scripts are used to set up the environment for the lab—both are included in the project for the lab, along with a sample solution for each exercise.

If you need to start again, open and execute the clean-up script, followed by the set-up script; you will be back to a clean environment and can try again.

1. Ensure that the 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa$$w0rd**.
2. In the D:\Labfiles\Lab07\Starter folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.
4. At the command prompt, press **y**, and then press **Enter**.
5. 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.ssmssln.**
4. In Solution Explorer, expand **Queries**, and double-click **01 Setup.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press **Ctrl+Alt+L** on the keyboard.)
5. Click **Execute** from the toolbar. 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 on the **New Query** icon.
7. When the query window opens, Type **USE TempDB**, followed by **GO** on the next line, and then click **Execute** on the toolbar.
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 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 which 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.

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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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. When prompted press **y**, and then **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 click **Project/Solution**.
3. In the Open Project window, open the project D:\Labfiles\Lab08\Starter\Project\Project.ssmssln.
4. In Solution Explorer, expand the **Queries** folder, and then double-click **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard.)
5. When the query window opens, highlight the statement **USE TSQL**; and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
SELECT N'The unit price for the ' + productname + N' is ' + CAST(unitprice AS
NVARCHAR(10)) + N' $.' AS productdesc
FROM Production.Products;
```

7. 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.
8. 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**.

3. Observe the error message:

```
Conversion failed when converting the nvarchar value '67.89.01.23' to data type int.
```

   Because you want to retrieve rows without conversion errors and have a NULL for those that produce a conversion error, you can use the TRY_CONVERT function.

4. Modify the query to use the TRY_CONVERT function. The query should look like this:

```
SELECT
TRY_CONVERT(INT, REPLACE(REPLACE(REPLACE(REPLACE(phone, N'-', N''), N'(', ''), N')',
''), ' ', '')) AS phonenoasint
FROM Sales.Customers;
```

5. Highlight the written query and click **Execute**. Observe the result. The rows that could not be converted have a NULL.

Results: 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 the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL**; and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
IIF(country = N'Mexico' AND contacttitle = N'Owner', N'Target group', N'Other') AS
segmentgroup, custid, contactname
FROM Sales.Customers;
```

4. The IIF function was new in SQL Server 2012. It was added mainly to support migrations from Microsoft Access to SQL Server. You can use the CASE expression to achieve the same result.
5. Highlight the written query and click **Execute**.

▶ Task 2: Modify the T-SQL Statement to Mark Different Customers

1. In the query pane, type the following query after the task 2 description:

```
SELECT
IIF(contacttitle = N'Owner' OR region IS NOT NULL, N'Target group', N'Other') AS
segmentgroup,  custid,  contactname
FROM Sales.Customers;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Create Four Groups of Customers

1. In the query pane, type the following query after the task 3 description:

```
SELECT CHOOSE(custid % 4 + 1, N'Group One', N'Group Two', N'Group Three', N'Group
Four') AS segmentgroup, custid, contactname
FROM Sales.Customers;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should know how to use the logical functions.

Exercise 3: Writing Queries That Test for Nullability

▶ Task 1: Write a SELECT Statement to Retrieve the Customer Fax Information

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL**; and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT contactname, COALESCE(fax, N'No information') AS faxinformation
FROM Sales.Customers;
```

This query uses the COALESCE function to retrieve customers' fax information.

4. Highlight the written query and click **Execute**.
5. In the query pane, type the following query after the previous query:

```
SELECT contactname, ISNULL(fax, N'No information') AS faxinformation
FROM Sales.Customers;
```

   This query uses the ISNULL function. What is the difference between the ISNULL and COALESCE functions? COALESCE is a standard ANSI SQL function and ISNULL is not. So you should use the COALESCE function.

6. Highlight the written query and click **Execute**.

▶ Task 2: Write a Filter for a Variable That Could Be a Null

1. Highlight the query provided under the task 2 description and click **Execute**.
2. Highlight the previous query. On the toolbar, click **Edit**, and then **Copy**.
3. In the query window, click the line after the task 2 description. On the toolbar, click **Edit** and then **Paste**. You have now copied the previous query to the same query window after the task 2 description.
4. Modify the query so that it looks like this:

```
DECLARE @region AS NVARCHAR(30) = NULL;
SELECT
custid, region
FROM Sales.Customers
WHERE region = @region OR (region IS NULL AND @region IS NULL);
```

5. Highlight the modified query and click **Execute**.

▶ 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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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 click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab09\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard).
5. When the query window opens, highlight the statement **USE TSQL;** and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 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** (or press F5 on the keyboard)**.**

▶ Task 3: Add an Additional Column From the Sales.Customers Table

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the task 3 description. On the toolbar, click **Edit**, and then **Paste**.
3. Modify the T-SQL statement so that it adds an additional column. Your query should look like this:

```
SELECT
o.custid, c.contactname, c.city
FROM Sales.Orders AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE o.empid = 5
GROUP BY o.custid, c.contactname;
```

4. Highlight the written query and click **Execute**.

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**.

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 the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
o.orderid, o.orderdate, SUM(d.qty * d.unitprice) AS salesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.orderid, o.orderdate
ORDER BY salesamount DESC;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Add Additional Columns

1. Highlight the previous query. On the toolbar, click **Edit**, and then **Copy**.
2. In the query window, click the line after the task 2 description. On the toolbar, click **Edit**, and then **Paste**.
3. Modify the T-SQL statement so that it adds extra columns. Your query should look like this:

```
SELECT
o.orderid, o.orderdate,
SUM(d.qty * d.unitprice) AS salesamount,
COUNT(*) AS noofoderlines,
AVG(d.qty * d.unitprice) AS avgsalesamountperorderline
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.orderid, o.orderdate
ORDER BY salesamount DESC;
```

4. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve the Sales Amount Value Per Month

1. In the query pane, type the following query after the task 3 description:

```
SELECT
YEAR(orderdate) * 100 + MONTH(orderdate) AS yearmonthno,
SUM(d.qty * d.unitprice) AS saleamountpermonth
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY YEAR(orderdate), MONTH(orderdate)
ORDER BY yearmonthno;
```

2. Highlight the written query and click **Execute**.

▶ Task 4: Write a SELECT Statement to List All Customers with the Total Sales Amount and Number of Order Lines Added

1. In the query pane, type the following query after the task 4 description:

```
SELECT
c.custid, c.contactname,
SUM(d.qty * d.unitprice) AS totalsalesamount,
MAX(d.qty * d.unitprice) AS maxsalesamountperorderline,
COUNT(*) AS numberofrows,
COUNT(o.orderid) AS numberoforderlines
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON o.custid = c.custid
LEFT OUTER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY c.custid, c.contactname
ORDER BY totalsalesamount;
```

2. Highlight the written query and click **Execute**.
3. Observe the result. Notice that the values in the **numberofrows** and **numberoforderlines** columns are different. Why? All aggregate functions ignore NULLs except COUNT(*), which is why you received the value 1 for the **numberofrows** column. When you used the **orderid** column in the COUNT function, you received the value 0 because the **orderid** is NULL for customers without an order.

Exercise 3: Writing Queries That Use Distinct Aggregate Functions

▶ Task 1: Modify a SELECT Statement to Retrieve the Number of Customers

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. Highlight the provided T-SQL statement after the Task 1 description and click **Execute**.
4. Observe the result. Notice that the number of orders is the same as the number of customers. Why? You are using the aggregate COUNT function on the **orderid** and **custid** columns and, 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 the query **81 - Lab Exercise 4.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT TOP (10)
o.custid,
SUM(d.qty * d.unitprice) AS totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING SUM(d.qty * d.unitprice) > 10000
ORDER BY totalsalesamount DESC;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Write a SELECT Statement to Retrieve Specific Orders

1. In the query pane, type the following query after the task 2 description:

```
SELECT
o.orderid,
o.empid,
SUM(d.qty * d.unitprice) as totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.orderdate >= '20080101' AND o.orderdate < '20090101'
GROUP BY o.orderid, o.empid;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Apply Additional Filtering

1. Highlight the previous query. On the toolbar, click **Edit**, and then **Copy**.
2. In the query window, click the line after the task 3 description. On the toolbar, click **Edit**, and then **Paste**.
3. Modify the T-SQL statement to apply additional filtering. Your query should look like this:

```
SELECT
o.orderid,
o.empid,
SUM(d.qty * d.unitprice) as totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.orderdate >= '20080101' AND o.orderdate < '20090101'
GROUP BY o.orderid, o.empid
HAVING SUM(d.qty * d.unitprice) >= 10000;
```

4. Highlight the written query and click **Execute**.
5. Modify the T-SQL statement to include an additional filter to retrieve only orders handled by the employee whose ID is 3. Your query should look like this:

```
SELECT
o.orderid,
o.empid,
SUM(d.qty * d.unitprice) as totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE
o.orderdate >= '20080101' AND o.orderdate <= '20090101'
AND o.empid = 3
GROUP BY o.orderid, o.empid
HAVING SUM(d.qty * d.unitprice) >= 10000;
```

   In this query, the predicate logic is applied in the WHERE clause. You could also write the predicate logic inside the HAVING clause. Which do you think is better?

   Unlike with **orderdate** filtering, with **empid** filtering, the result is going to be correct either way because you are filtering by an element that appears in the GROUP BY list. Conceptually, it seems more intuitive to filter as early as possible. This query then applies the filtering in the WHERE clause because it will be logically applied before the GROUP BY clause. Do not forget, though, that the actual processing in the SQL Server engine could be different.

6. Highlight the written query and click **Execute**.

▶ Task 4: Retrieve the Customers with More Than 25 Orders

1. In the query pane, type the following query after the task 4 description:

```
SELECT
o.custid,
MAX(orderdate) AS lastorderdate,
SUM(d.qty * d.unitprice) AS totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING COUNT(DISTINCT o.orderid) > 25;
```

2. Highlight the written query and click **Execute**.

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

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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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. When prompted, 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** window, open the project **D:\Labfiles\Lab10\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and double-click **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, select **Solution Explorer** on the **View** menu or press Ctrl+Alt+L on the keyboard.)
5. When the query window opens, highlight the statement **USE TSQL**; and click **Execute** on the toolbar (or press F5 on the keyboard).
6. In the query pane, type the following query after the task 1 description:

```
SELECT MAX(orderdate) AS lastorderdate
FROM Sales.Orders;
```

7. Highlight the written query and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve All Orders Placed on the Last Order Date

1. In the query pane, type the following query after the task 2 description:

```
SELECT
orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE
orderdate = (SELECT MAX(orderdate) FROM Sales.Orders);
```

2. Highlight the written query and click **Execute**.

▶ Task 4: Observe the T-SQL Statement Provided by the IT Department

1. Highlight the provided T-SQL statement under the task 3 description and click **Execute**.
2. Modify the query to filter customers whose contact name starts with the letter B. Your query should look like this:

```
SELECT
orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE
custid =
(
SELECT custid
FROM Sales.Customers
WHERE contactname LIKE N'B%'
);
```

3. Highlight the written query and click **Execute**.
4. Observe the error message:

```
Subquery returned more than 1 value. This is not permitted when the subquery follows
=, !=, <, <= , >, >= or when the subquery is used as an expression.
```

   Why did the query fail? It failed because the subquery returned more than one row. To fix this problem, you should replace the = operator with an IN operator.

5. Modify the query so that it uses the IN operator. Your query should look like this:

```
SELECT
orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE
custid IN
(
SELECT custid
FROM Sales.Customers
WHERE contactname LIKE N'B%'
);
```

6. Highlight the written query and click **Execute**.

▶ Task 5: Write A SELECT Statement to Analyze Each Order's Sales as a Percentage of the Total Sales Amount

1. In the query pane, type the following query after the task 4 description:

```
SELECT
o.orderid,
SUM(d.qty * d.unitprice) AS totalsalesamount,
SUM(d.qty * d.unitprice) /
(
SELECT SUM(d.qty * d.unitprice)
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.orderdate >= '20080501' AND orderdate < '20080601'
) * 100. AS salespctoftotal
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.orderdate >= '20080501' AND orderdate < '20080601'
GROUP BY o.orderid;
```

2. Highlight the written query and click **Execute.**

Results: After this exercise, you should be able to use self-contained subqueries in T-SQL statements.

Exercise 2: Writing Queries That Use Scalar and Multi-Result Subqueries

▶ Task 1: Write a SELECT Statement to Retrieve Specific Products

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL**; and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
productid, productname
FROM Production.Products
WHERE
productid IN
(
SELECT productid
FROM Sales.OrderDetails
WHERE qty > 100
);
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Write a SELECT Statement to Retrieve Those Customers Without Orders

1. In the query pane, type the following query after the task 2 description:

```
SELECT
custid, contactname
FROM Sales.Customers
WHERE custid NOT IN
(
SELECT custid
FROM Sales.Orders
);
```

2. Highlight the written query and click **Execute**.
3. Observe the result. Notice there are two customers without an order.

▶ Task 3: Add a Row and Rerun the Query That Retrieves Those Customers Without Orders

1. Highlight the provided T-SQL statement under the task 3 description and click **Execute**. This code inserts an additional row that has a NULL in the **custid** column of the **Sales.Orders** table.
2. Highlight the query in task 2. On the toolbar, click **Edit** and then **Copy**.
3. In the query window, click the line after the provided T-SQL statement. On the toolbar, click **Edit** and then **Paste**.
4. Highlight the written query and click **Execute**.

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 multi-result subqueries in T-SQL statements.

Exercise 3: Writing Queries That Use Correlated Subqueries and an EXISTS Predicate

▶ Task 1: Write a SELECT Statement to Retrieve the Last Order Date for Each Customer

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL**; and click **Execute**.
3. In the query pane, type the following query after the task 1 description:

```
SELECT
c.custid, c.contactname,
(
SELECT MAX(o.orderdate)
FROM Sales.Orders AS o
WHERE o.custid = c.custid
) AS lastorderdate
FROM Sales.Customers AS c;
```

4. Highlight the written query and click **Execute**.

▶ 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

1. Under the task 5 description, highlight the provided T-SQL statement and click **Execute**.

Results: After this exercise, you should have an understanding of how to use a correlated subquery in T-SQL statements.

Module 11: 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 20761A-MIA-DC and 20761A-MIA-SQL virtual machines are both running, and then log on to 20761A-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**, 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 click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab11\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand the **Queries** folder, and then double-click the query **51 - Lab Exercise 1.sql**. (If Solution Explorer is not visible, on the **View** menu, click **Solution Explorer**.
5. In the query pane, highlight the statement **USE TSQL;** and 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. On the **Edit** menu, click **Copy**.
5. In the query window, click the line after the written T-SQL statement. 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 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, 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. On the menu **Edit**, click **Copy**.
2. In the query window, click the line after the task 3 description. 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. On the **Edit** menu, click **Copy**.
2. In the query window, click the line after the task 4 description. 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

1. Highlight the provided T-SQL statement after the task 5 description, and click **Execute**.

Results: After this exercise, you should be able to use the CROSS APPLY and OUTER APPLY operators in your T-SQL statements.

Exercise 3: Writing Queries That Use the EXCEPT and INTERSECT Operators

▶ Task 1: Write a SELECT Statement to Return All Customers Who Bought More Than 20 Distinct Products

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;**, and click **Execute**.
3. In the query pane, 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;
```

4. Highlight the written query, and click **Execute**.

▶ Task 2: Write a SELECT Statement to Retrieve All Customers from the USA, Except Those Who Bought More Than 20 Distinct Products

1. In the query pane, 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;
```

2. Highlight the written query, and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve Customers Who Spent More Than $10,000

1. In the query pane, 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. On **Edit** menu, click **Copy**.
2. In the query window, click the line after the task 4 description. 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. On the **Edit** menu, click **Copy**.
2. In the query window, click the line after the task 5 description. 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.

Results: After this exercise, you should have an understanding of how to use the EXCEPT and INTERSECT operators in T-SQL statements.