



OFFICIAL MICROSOFT LEARNING PRODUCT

20762C

Developing SQL Databases

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

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

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

© 2017 Microsoft Corporation. All rights reserved.

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

Product Number: 20762C

Part Number (if applicable): X21-64455

Released: 12/2017

MICROSOFT LICENSE TERMS

MICROSOFT INSTRUCTOR-LED COURSEWARE

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

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

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

1. DEFINITIONS.

- a. "Authorized Learning Center" means a Microsoft IT Academy Program Member, Microsoft Learning Competency Member, or such other entity as Microsoft may designate from time to time.
- b. "Authorized Training Session" means the instructor-led training class using Microsoft Instructor-Led Courseware conducted by a Trainer at or through an Authorized Learning Center.
- c. "Classroom Device" means one (1) dedicated, secure computer that an Authorized Learning Center owns or controls that is located at an Authorized Learning Center's training facilities that meets or exceeds the hardware level specified for the particular Microsoft Instructor-Led Courseware.
- d. "End User" means an individual who is (i) duly enrolled in and attending an Authorized Training Session or Private Training Session, (ii) an employee of a MPN Member, or (iii) a Microsoft full-time employee.
- e. "Licensed Content" means the content accompanying this agreement which may include the Microsoft Instructor-Led Courseware or Trainer Content.
- f. "Microsoft Certified Trainer" or "MCT" means an individual who is (i) engaged to teach a training session to End Users on behalf of an Authorized Learning Center or MPN Member, and (ii) currently certified as a Microsoft Certified Trainer under the Microsoft Certification Program.
- g. "Microsoft Instructor-Led Courseware" means the Microsoft-branded instructor-led training course that educates IT professionals and developers on Microsoft technologies. A Microsoft Instructor-Led Courseware title may be branded as MOC, Microsoft Dynamics or Microsoft Business Group courseware.
- h. "Microsoft IT Academy Program Member" means an active member of the Microsoft IT Academy Program.
- i. "Microsoft Learning Competency Member" means an active member of the Microsoft Partner Network program in good standing that currently holds the Learning Competency status.
- j. "MOC" means the "Official Microsoft Learning Product" instructor-led courseware known as Microsoft Official Course that educates IT professionals and developers on Microsoft technologies.
- k. "MPN Member" means an active Microsoft Partner Network program member in good standing.

- I. "Personal Device" means one (1) personal computer, device, workstation or other digital electronic device that you personally own or control that meets or exceeds the hardware level specified for the particular Microsoft Instructor-Led Courseware.
 - m. "Private Training Session" means the instructor-led training classes provided by MPN Members for corporate customers to teach a predefined learning objective using Microsoft Instructor-Led Courseware. These classes are not advertised or promoted to the general public and class attendance is restricted to individuals employed by or contracted by the corporate customer.
 - n. "Trainer" means (i) an academically accredited educator engaged by a Microsoft IT Academy Program Member to teach an Authorized Training Session, and/or (ii) a MCT.
 - o. "Trainer Content" means the trainer version of the Microsoft Instructor-Led Courseware and additional supplemental content designated solely for Trainers' use to teach a training session using the Microsoft Instructor-Led Courseware. Trainer Content may include Microsoft PowerPoint presentations, trainer preparation guide, train the trainer materials, Microsoft One Note packs, classroom setup guide and Pre-release course feedback form. To clarify, Trainer Content does not include any software, virtual hard disks or virtual machines.
- 2. USE RIGHTS.** The Licensed Content is licensed not sold. The Licensed Content is licensed on a ***one copy per user basis***, such that you must acquire a license for each individual that accesses or uses the Licensed Content.

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

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

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

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 performance¹. Ask your instructor about Microsoft Certification exam promotions and discounts that may be available to you.
- **Customer Satisfaction Guarantee**—Our Certified Partners for Learning Solutions offer a satisfaction guarantee and we hold them accountable for it. At the end of class, please complete an evaluation of today's experience. We value your feedback!

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

Sincerely,

Microsoft Learning
www.microsoft.com/learning



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

Acknowledgements

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

Aaron Johal – Content Developer

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

Alistair Matthews – Content Developer

Alistair Matthews is an experienced consultant and technical author who has worked with a wide variety of Microsoft technologies over his nineteen-year career. Recent projects have included revising SQL Server 2014 courseware, authorizing Azure courseware, and developing SharePoint apps, all for different groups within Microsoft. He designed and wrote the Microsoft Official Curriculum course number 20486 "Developing ASP.NET 4.5 MVC Web Applications" and several modules in the SharePoint developers' courses numbers 20488 and 20489. He has developed content for Microsoft Press books, TechNet, and MSDN and has written e-learning courseware and examination questions. He lives in Cornwall in the United Kingdom.

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.

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.

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: An Introduction to Database Development

Module Overview	1-1
Lesson 1: Introduction to the SQL Server Platform	1-2
Lesson 2: SQL Server Database Development Tasks	1-11
Module Review and Takeaways	1-17

Module 2: Designing and Implementing Tables

Module Overview	2-1
Lesson 1: Designing Tables	2-2
Lesson 2: Data Types	2-12
Lesson 3: Working with Schemas	2-26
Lesson 4: Creating and Altering Tables	2-31
Lab: Designing and Implementing Tables	2-38
Module Review and Takeaways	2-41

Module 3: Advanced Table Designs

Module Overview	3-1
Lesson 1: Partitioning Data	3-2
Lesson 2: Compressing Data	3-13
Lesson 3: Temporal Tables	3-19
Lab: Using Advanced Table Designs	3-27
Module Review and Takeaways	3-31

Module 4: Ensuring Data Integrity Through Constraints

Module Overview	4-1
Lesson 1: Enforcing Data Integrity	4-2
Lesson 2: Implementing Data Domain Integrity	4-6
Lesson 3: Implementing Entity and Referential Integrity	4-11
Lab: Ensuring Data Integrity Through Constraints	4-22
Module Review and Takeaways	4-25

Module 5: Introduction to Indexes

Module Overview	5-1
Lesson 1: Core Indexing Concepts	5-2
Lesson 2: Data Types and Indexes	5-8
Lesson 3: Heaps, Clustered, and Nonclustered Indexes	5-12
Lesson 4: Single Column and Composite Indexes	5-22
Lab: Implementing Indexes	5-26
Module Review and Takeaways	5-29

Module 6: Designing Optimized Index Strategies

Module Overview	6-1
Lesson 1: Index Strategies	6-2
Lesson 2: Managing Indexes	6-7
Lesson 3: Execution Plans	6-16
Lesson 4: The Database Engine Tuning Advisor	6-25
Lesson 5: Query Store	6-27
Lab: Optimizing Indexes	6-34
Module Review and Takeaways	6-37

Module 7: Columnstore Indexes

Module Overview	7-1
Lesson 1: Introduction to Columnstore Indexes	7-2
Lesson 2: Creating Columnstore Indexes	7-7
Lesson 3: Working with Columnstore Indexes	7-12
Lab: Using Columnstore Indexes	7-17
Module Review and Takeaways	7-21

Module 8: Designing and Implementing Views

Module Overview	8-1
Lesson 1: Introduction to Views	8-2
Lesson 2: Creating and Managing Views	8-9
Lesson 3: Performance Considerations for Views	8-18
Lab: Designing and Implementing Views	8-22
Module Review and Takeaways	8-25

Module 9: Designing and Implementing Stored Procedures

Module Overview	9-1
Lesson 1: Introduction to Stored Procedures	9-2
Lesson 2: Working with Stored Procedures	9-7
Lesson 3: Implementing Parameterized Stored Procedures	9-16
Lesson 4: Controlling Execution Context	9-21
Lab: Designing and Implementing Stored Procedures	9-25
Module Review and Takeaways	9-29

Module 10: Designing and Implementing User-Defined Functions

Module Overview	10-1
Lesson 1: Overview of Functions	10-2
Lesson 2: Designing and Implementing Scalar Functions	10-5
Lesson 3: Designing and Implementing Table-Valued Functions	10-10
Lesson 4: Considerations for Implementing Functions	10-14
Lesson 5: Alternatives to Functions	10-20
Lab: Designing and Implementing User-Defined Functions	10-22
Module Review and Takeaways	10-24

Module 11: Responding to Data Manipulation Via Triggers

Module Overview	11-1
Lesson 1: Designing DML Triggers	11-2
Lesson 2: Implementing DML Triggers	11-9
Lesson 3: Advanced Trigger Concepts	11-15
Lab: Responding to Data Manipulation by Using Triggers	11-23
Module Review and Takeaways	11-26

Module 12: Using In-Memory Tables

Module Overview	12-1
Lesson 1: Memory-Optimized Tables	12-2
Lesson 2: Natively Compiled Stored Procedures	12-11
Lab: Using In-Memory Database Capabilities	12-16
Module Review and Takeaways	12-19

Module 13: Implementing Managed Code in SQL Server

Module Overview	13-1
Lesson 1: Introduction to CLR Integration in SQL Server	13-2
Lesson 2: Implementing and Publishing CLR Assemblies	13-9
Lab: Implementing Managed Code in SQL Server	13-17
Module Review and Takeaways	13-20

Module 14: Storing and Querying XML Data in SQL Server

Module Overview	14-1
Lesson 1: Introduction to XML and XML Schemas	14-2
Lesson 2: Storing XML Data and Schemas in SQL Server	14-11
Lesson 3: Implementing the XML Data Type	14-18
Lesson 4: Using the Transact-SQL FOR XML Statement	14-22
Lesson 5: Getting Started with XQuery	14-34
Lesson 6: Shredding XML	14-42
Lab: Storing and Querying XML Data in SQL Server	14-52
Module Review and Takeaways	14-57

Module 15: Storing and Querying Spatial Data in SQL Server

Module Overview	15-1
Lesson 1: Introduction to Spatial Data	15-2
Lesson 2: Working with SQL Server Spatial Data Types	15-7
Lesson 3: Using Spatial Data in Applications	15-15
Lab: Working with SQL Server Spatial Data	15-20
Module Review and Takeaways	15-23

Module 16: Storing and Querying BLOBs and Text Documents in SQL Server

Module Overview	16-1
Lesson 1: Considerations for BLOB Data	16-2
Lesson 2: Working with FILESTREAM	16-9
Lesson 3: Using Full-Text Search	16-16
Lab: Storing and Querying BLOBs and Text Documents in SQL Server	16-26
Module Review and Takeaways	16-30

Module 17: SQL Server Concurrency

Module Overview	17-1
Lesson 1: Concurrency and Transactions	17-2
Lesson 2: Locking Internals	17-14
Lab: Concurrency and Transactions	17-27
Module Review and Takeaways	17-31

Module 18: Performance and Monitoring

Module Overview	18-1
Lesson 1: Extended Events	18-2
Lesson 2: Working with Extended Events	18-11
Lesson 3: Live Query Statistics	18-20
Lesson 4: Optimize Database File Configuration	18-23
Lesson 5: Metrics	18-27
Lab: Monitoring, Tracing, and Baselining	18-37
Module Review and Takeaways	18-40

Lab Answer Keys

Module 1 Lab: Designing and Implementing Tables	L01-1
Module 2 Lab: Using Advanced Table Designs	L02-1
Module 3 Lab: Ensuring Data Integrity Through Constraints	L03-1
Module 4 Lab: Implementing Indexes	L04-1
Module 5 Lab: Optimizing Indexes	L05-1
Module 6 Lab: Using Columnstore Indexes	L06-1
Module 7 Lab: Designing and Implementing Views	L07-1
Module 8 Lab: Designing and Implementing Stored Procedures	L08-1
Module 9 Lab: Designing and Implementing User-Defined Functions	L09-1
Module 10 Lab: Responding to Data Manipulation by Using Triggers	L10-1
Module 11 Lab: Using In-Memory Database Capabilities	L11-1
Module 12 Lab: Implementing Managed Code in SQL Server	L12-1
Module 13 Lab: Storing and Querying XML Data in SQL Server	L13-1
Module 14 Lab: Working with SQL Server Spatial Data	L14-1
Module 15 Lab: Storing and Querying BLOBS and Text Documents in SQL Server	L15-1
Module 16 Lab: Concurrency and Transactions	L16-1
Module 17 Lab: Monitoring, Tracing, and Baselining	L17-1

About This Course

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

Course Description



Note: This release of course 20762 supersedes 20762B, and is based on SQL Server 2017

This five-day instructor-led course provides students with the knowledge and skills to develop a Microsoft SQL Server database. The course focuses on teaching individuals how to use SQL Server product features and tools related to developing a database.

Audience

The primary audience for this course is IT Professionals who want to become skilled on SQL Server product features and technologies for implementing a database.

The secondary audiences for this course are individuals who are developers from other product platforms looking to become skilled in the implementation of a SQL Server database.

Student Prerequisites

In addition to their professional experience, students who attend this training should already have the following technical knowledge:

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

Course Objectives

After completing this course, students will be able to:

- Design and Implement Tables.
- Describe advanced table designs
- Ensure Data Integrity through Constraints.
- Describe indexes, including Optimized and Columnstore indexes
- Design and Implement Views.
- Design and Implement Stored Procedures.
- Design and Implement User Defined Functions.
- Respond to data manipulation using triggers.
- Design and Implement In-Memory Tables.
- Implement Managed Code in SQL Server.
- Store and Query XML Data.
- Work with Spatial Data.
- Store and Query Blobs and Text Documents.
- Measure and Monitor Performance

Course Outline

The course outline is as follows:

- Module 1: 'Introduction to database development' introduces the entire SQL Server platform and its major tools. It will cover editions, versions, basics of network listeners, and concepts of services and service accounts.
- Module 2: 'Designing and implementing tables' describes the design and implementation of tables. (Note: partitioned tables are not covered).
- Module 3: 'Advanced table designs' describes more advanced table designs.
- Module 4: 'Ensuring data integrity through constraints' describes the design and implementation of constraints.
- Module 5: 'Introduction to indexes' describes the concept of an index and discusses selectivity, density and statistics. It covers appropriate data type choices and choices around composite index structures.
- Module 6: 'Designing optimized index strategies' includes covering indexes and the INCLUDE clause, hints, padding / fillfactor, statistics. It also covers execution plans and the DTE Lessons.
- Module 7: 'Columnstore indexes' introduces Columnstore indexes.
- Module 8: 'Designing and implementing views' describes the design and implementation of views.
- Module 9: 'Designing and implementing stored procedures' describes the design and implementation of stored procedures.
- Module 10: 'Designing and implementing user-defined functions' describes the design and implementation of functions, both scalar and table-valued. (Also discusses where they can lead to performance issues).
- Module 11: 'Responding to data-manipulation via triggers' describes the design and implementation of triggers.
- Module 12: 'Using in-memory tables' covers the creation of in-memory tables and native stored procedures. Furthermore, advantages of in-memory tables are discussed, for example the removal of transaction blocking.
- Module 13: 'Implementing managed code in SQL server' describes the implementation of and target use-cases for SQL CLR integration.
- Module 14: 'Storing and querying XML data in SQL server' covers the XML data type, schema collections, typed and un-typed columns and appropriate use cases for XML in SQL Server.
- Module 15: 'Storing and Querying Spatial Data in SQL Server' describes spatial data and how this data can be implemented within SQL Server.
- Module 16: 'Storing and querying blobs and text documents in SQL server' covers full text indexes and queries.
- Module 17: 'SQL Server Concurrency' covers what concurrency is, why it is important, and the different ways you can configure and work with concurrency settings.
- Module 18: 'Performance and Monitoring' looks at how to measure and monitor performance of your SQL Server databases.

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, send email to mcspprt@microsoft.com. 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
20762C-MIA-DC	MIA-DC1 is a domain controller.
20762C-MIA-SQL	MIA-SQL has SQL Server 2017 installed
MSL-TMG1	TMG1 is used to access the internet

Software Configuration

The following software is installed on the virtual machines:

- Windows Server 2016
- SQL2017
- SharePoint 2016

Course Files

The files associated with the labs in this course are located in the D:\Labfiles folder on the 20762C-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 Learning Partner classrooms in which Official Microsoft Learning Product courseware is taught.

- Intel Virtualization Technology (Intel VT) or AMD Virtualization (AMD-V) processor
- Dual 120-gigabyte (GB) hard disks 7200 RPM Serial ATA (SATA) or better
- 16 GB of random access memory (RAM)
- DVD drive
- Network adapter
- Super VGA (SVGA) 17-inch monitor
- Microsoft mouse or compatible pointing device

- Sound card with amplified speakers

Additionally, the instructor's computer must be connected to a projection display device that supports SVGA 1024×768 pixels, 16-bit colors.

Module 1

An Introduction to Database Development

Contents:

Module Overview	1-1
Lesson 1: Introduction to the SQL Server Platform	1-2
Lesson 2: SQL Server Database Development Tasks	1-11
Module Review and Takeaways	1-17

Module Overview

Before beginning to work with Microsoft® SQL Server® in either a development or an administration role, it is important to understand the scope of the SQL Server platform. In particular, it is useful to understand that SQL Server is not just a database engine—it is a complete platform for managing enterprise data.

SQL Server provides a strong data platform for all sizes of organizations, in addition to a comprehensive set of tools to make development easier, and more robust.

Objectives

After completing this module, you will be able to:

- Describe the SQL Server platform.
- Understand the common tasks undertaken during database development and the SQL Server Tools provided to support these tasks.

Lesson 1

Introduction to the SQL Server Platform

Microsoft SQL Server data management software is a platform for developing business applications that are data focused. Rather than being a single, monolithic application, SQL Server is structured as a series of components. It is important to understand the use of each component.

You can install more than one copy of SQL Server on a server. Each copy is called an instance and you can configure and manage them separately.

There are various editions of SQL Server, and each one has a different set of capabilities. It is important to understand the target business cases for each, and how SQL Server has evolved through a series of improving versions over many years. It is a stable and robust data management platform.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the overall SQL Server platform.
- Explain the role of each of the components that make up the SQL Server platform.
- Describe the functionality that SQL Server instances provide.
- Explain the available SQL Server editions.
- Explain how SQL Server has evolved through a series of versions.

SQL Server Architecture

The SQL Server Database Engine is one component in the suite of products that comprise SQL Server. However, the database engine is not a homogenous piece of software; it is made up of different modules, each with a separate function.

SQL Server Operating System

Underpinning SQL Server Database Engine is the SQL Server Operating System (SQLOS). This performs functions such as managing memory, managing locks, the thread scheduler, the buffer pool, and much more. It also provides access to external components, such as the common language runtime (CLR).

If you want to explore the SQLOS in more detail, look at the list of dynamic management views (DMVs) related to the SQLOS. A full list is provided in Microsoft Docs:



SQL Server Operating System Related Dynamic Management Views (Transact-SQL)

<http://aka.ms/Ye1hmt>

- SQL Server architecture:
 - SQL Server Operating System (SQLOS)
 - Database Engine
 - Query processor
- Complete set of enterprise-ready technologies and tools
- Low total cost of ownership
- Highly integrated platform

Database Engine

The SQL Server Database Engine is made of up two main parts:

1. The storage engine.
2. The query processor.

The storage engine manages access to data stored in the database, including how the data is physically stored on disk, backups and restores, indexes, and more.

The query processor ensures that queries are formatted correctly; it plans how best to execute a query, and then executes the query.

SQL Server is an integrated and enterprise-ready platform for data management that offers a low total cost of ownership.

Enterprise Ready

SQL Server provides a very secure, robust, and stable relational database management system, but there is much more to it than that. You can use SQL Server to manage organizational data and provide both analysis of, and insights into, that data. Microsoft also provides other enterprise development environments—for example, Visual Studio®—that have excellent integration and support for SQL Server.

The SQL Server Database Engine is one of the highest performing database engines available and regularly features in the top tier of industry performance benchmarks. You can review industry benchmarks and scores on the Transaction Processing Performance Council (TPC) website.

High Availability

Impressive performance is necessary, but not at the cost of availability. Many enterprises are now finding it necessary to provide access to their data 24 hours a day, seven days a week. The SQL Server platform was designed with the highest levels of availability in mind. As each version of the product has been released, more capabilities have been added to minimize any potential downtime.

Security

Uppermost in the minds of enterprise managers is the requirement to secure organizational data. It is not possible to retrofit security after an application or product has been created. From the outset, SQL Server has been built with the highest levels of security as a goal. SQL Server includes encryption features such as Always Encrypted, designed to protect sensitive data such as credit card numbers, and other personal information.

Scalability

Organizations require data management capabilities for systems of all sizes. SQL Server scales from the smallest needs, running on a single desktop, to the largest—a high-availability server farm—via a series of editions that have increasing capabilities.

Cost of Ownership

Many competing database management systems are expensive both to purchase and to maintain. SQL Server offers very low total cost of ownership. SQL Server tooling (both management and development) builds on existing Windows® knowledge. Most users tend to quickly become familiar with the tools. The productivity users can achieve when they use the various tools is enhanced by the high degree of integration between them. For example, many of the SQL Server tools have links to launch and preconfigure other SQL Server tools.

SQL Server Components

SQL Server is an excellent relational database engine, but as a data platform, it offers much more than this. SQL Server consists of several components as described in the following table:

- Components
 - Database Engine
 - SQL Server Analysis Services, Reporting Services, and Integration Services.
 - Master Data Services and Data Quality Services
- Tools
 - SQL Server Management Studio and Data Tools
 - Configuration Manager
 - Profiler
 - Tuning Advisor
 - DQS Client

Component	Description
SQL Server Database Engine	A relational database engine based on Structured Query Language (SQL), the core service for storing, processing, and securing data, replication, full-text search, tools for managing relational and XML data.
Analysis Services (SSAS)	An online analytical processing (OLAP) engine that works with analytic cubes and supports data mining applications.
Reporting Services (SSRS)	Offers a reporting engine based on web services and provides a web portal and end-user reporting tools. It is also an extensible platform that you can use to develop report applications.
Integration Services (SSIS)	Used to orchestrate the movement of data between SQL Server components and other external systems. Traditionally used for extract, transform and load (ETL) operations.
Master Data Services (MDS)	Provides tooling for managing master or reference data and includes hierarchies, granular security, transactions, data versioning, and business rules, as well as an add-in for Excel® for managing data.
Data Quality Services (DQS)	With DQS, you can build a knowledge base and use it to perform data quality tasks, including correction, enrichment, standardization, and de-duplication of data.
Replication	A set of technologies for copying and distributing data and database objects between multiple databases.

Alongside these server components, SQL Server provides the following management tools:

Tool	Description
SQL Server Management Studio (SSMS)	An integrated data management environment designed for developers and database administrators to manage the core database engine.
SQL Server Configuration Manager	Provides basic configuration management of services, client and server protocols, and client aliases.
SQL Server Profiler	A graphical user interface to monitor and assist in the management of performance of database engine and Analysis Service components.
Database Engine Tuning Advisor	Provides guidance on, and helps to create, the optimal sets of indexes, indexed views, and partitions.
Data Quality Services Client	A graphical user interface that connects to a DQS server, and then provides data cleansing operations and monitoring of their performance.
SQL Server Data Tools (SSDT)	An integrated development environment for developing business intelligence (BI) solutions utilizing SSAS, SSRS, and SSIS.
Connectivity Components	Components that facilitate communication between clients and servers. For example, ODBC, and OLE DB.

SQL Server Instances

It is sometimes useful to install more than one copy of a SQL Server Database Engine on a single server. You can install many instances, but the first instance will become the default. All other instances should then be named. Some components can be installed once, and used by multiple instances of the database engine, such as SQL Server Data Tools (SSDT).

- SQL Server components are instance-aware
- Allow for a degree of isolation
- Can assist in upgrade situations
- Two instance types:
 - Default instance
 - Named instance

Multiple Instances

The ability to install multiple instances of SQL Server components on a single server is useful in several situations:

- You may require different administrators or security environments for sets of databases. Each instance of SQL Server can be managed and secured separately.
- Applications may require SQL Server configurations that are inconsistent or incompatible with the server requirements of other applications. Each instance of SQL Server can be configured independently.
- You might want to support application databases with different levels of service, particularly in relation to availability. To meet different service level agreements (SLAs), you can create SQL Server instances to separate workloads.
- You might want to support different versions or editions of SQL Server.
- Applications might require different server-level collations. Although each database can have a different collation, an application might be dependent on the collation of the **tempdb** database which is shared between all databases. In this case you can create separate instances, with each instance having the correct collation.

You can install different versions of SQL Server side by side, using multiple instances. This can assist when testing upgrade scenarios or performing upgrades.

Default and Named Instances

One instance can be the default instance on a database server; this instance will have no name. Connection requests will connect to the default instance if it is sent to a computer without specifying an instance name. There is no requirement to have a default instance, because you can name every instance.

All other instances of SQL Server require an instance name, in addition to the server name, and are known as "named" instances. You cannot install all components of SQL Server in more than one instance. A substantial change in SQL Server 2012 was the introduction of multiple instance support for SQL Server Integration Services (SSIS).

You do not have to install SSDT more than once. A single installation of the tools can manage and configure all installed instances.

SQL Server Editions

SQL Server is available in a wide variety of editions. These have different price points and different levels of capability.

Use Cases for SQL Server Editions

Each SQL Server edition is targeted to a specific business use case, as shown in the following table:

- SQL Server Editions
 - Enterprise
 - Standard
 - Web
 - Developer
 - Express
 - Plus
 - Azure SQL Database

Edition	Business Use Case
Enterprise	Provides comprehensive high-end datacenter capabilities and includes end-to-end BI.
Standard	Provides basic data management and BI capabilities.
Developer	Includes all the capabilities of the Enterprise edition, licensed to be used in development and test environments. It cannot be used as a production server.
Express	Free database that supports learning and building small desktop data-driven applications.
Web	Gives a low total-cost-of-ownership option to provide database capabilities to small and large scale web properties.
Azure® SQL Database	Helps you to build database applications on a scalable and robust cloud platform. This is the Azure version of SQL Server.

SQL Server Versions

SQL Server has a rich history of innovation that has been achieved while maintaining strong levels of stability. It has been available for many years, yet it is still rapidly evolving, with new capabilities and features. Indeed, this evolution has begun to increase in speed, with Microsoft supporting SQL Server in the cloud with Azure.

Early Versions

The earliest versions of SQL Server (1.0 and 1.1) were based on the OS/2 operating system.

Versions 4.2 and later moved to the Windows operating system, initially on the Windows NT operating system.

Later Versions

Version 7.0 saw a significant rewrite of the product. Substantial advances were made to reduce the administration workload for the product. OLAP Services, which later became Analysis Services, was introduced.

SQL Server 2000 featured support for multiple instances and collations. It also introduced support for data mining. SSRS was introduced after the product release as an add-on enhancement, along with support for 64-bit processors.

SQL Server 2005 provided support for non-relational data that was stored and queried as XML, and SSMS was released to replace several previous administrative tools. SSIS replaced a tool formerly known as Data Transformation Services (DTS). Dynamic management views (DMVs) and functions were introduced to provide detailed health monitoring, performance tuning, and troubleshooting. Also, substantial high-availability improvements were included in the product, and database mirroring was introduced.

SQL Server 2008 provided AlwaysOn technologies to reduce potential downtime. Database compression and encryption technologies were added. Specialized date-related and time-related data types were introduced, including support for time zones within date/time data. Full-text indexing was integrated directly within the database engine. (Previously, full-text indexing was based on interfaces to services at the operating system level.) Additionally, a Windows PowerShell® provider for SQL Server was introduced.

SQL Server 2008 R2 added substantial enhancements to SSRS—the introduction of advanced analytic capabilities with PowerPivot; support for managing reference data with the introduction of Master Data Services; and the introduction of StreamInsight, with which users could query data that was arriving at high speed, before storing the data in a database.

SQL Server 2012 introduced tabular data models into SSAS. New features included: an enhancement of FileTable called FileStream; Semantic Search, with which users could extract statistically relevant words; the ability to migrate BI projects into Microsoft Visual Studio 2010.

SQL Server 2014 included substantial performance gains from the introduction of in-memory tables and native stored procedures. It also increased integration with Microsoft Azure.

SQL Server 2016 was a major release and added three important security features: Always Encrypted, dynamic data masking, and row-level security. This version also included stretch database to archive data in Microsoft Azure, Query Store to maintain a history of execution plans, PolyBase to connect to Hadoop data, temporal tables, and support for R, plus in-memory enhancements and columnstore indexes.

Version	Compatibility Level	Oldest Supported Level
SQL Server 2000	80	70
SQL Server 2005	90	70
SQL Server 2008	100	80
SQL Server 2012	110	90
SQL Server 2014	120	100
SQL Server 2016	130	100
SQL Server 2017	140	100

Current Version

SQL Server 2017 includes many fixes and enhancements, including:

- **SQL Graph.** Enables many-to-many relationships to be modelled more easily. Extensions to Transact-SQL includes new syntax to create tables as EDGES or NODES, and the MATCH keyword for querying.
- **Adaptive query processing.** This is a family of features that help queries to run more efficiently. They are **batch mode adaptive joins**, **batch mode memory grant feedback**, and **interleaved execution** for multi-statement table valued functions..
- **Automatic database tuning.** This allows query performance to either be fixed automatically, or to provide insight into potential problems so that fixes can be applied.
- **In-memory enhancements.** This includes computed columns in memory-optimized tables, support for CROSS APPLY in natively compiled modules, and support for JSON functions in natively compiled modules.
- **SQL Server Analysis Services.** This includes several enhancements for tabular models.
- **Machine Learning.** R is now known as SQL Server Machine Learning and includes support for Python as well as R. SQL Server 2017 includes

Compatibility

Businesses can run different versions of databases on an instance of SQL Server. Each version of SQL Server can build and maintain databases created on previous versions of SQL Server. For example, SQL Server 2016 can read and create databases at compatibility level 100; that is, databases created on SQL Server 2008. The compatibility level specifies the supported features of the database. For more information on compatibility levels, see Microsoft Docs:

 **ALTER DATABASE Compatibility Level (Transact-SQL)**

<http://aka.ms/xrf36h>

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	Database Engine
2	Data Quality Services Client
3	Enterprise
4	Master Data Services
5	Connectivity
6	Developer
7	Replication
8	Profiler
9	Web
10	Integration Services
11	SQL Server Management Studio
12	Standard
13	SQL Server Data Tools

Category 1	Category 2	Category 3
Component	Tool	Edition

Lesson 2

SQL Server Database Development Tasks

Microsoft provides numerous tools and integrated development environments to support database developers. This lesson investigates some of the common tasks undertaken by developers, how SQL Server supports those tasks, and which tools you can use to complete them.

Lesson Objectives

After completing this lesson, you will be able to:

- List the common tasks that a database developer undertakes.
- Describe the functionality of some of the SQL Server tools.
- Use SSMS and SSDT to connect to local and cloud-based databases.

Common Database Development Tasks

Developing software to solve business problems normally requires some form of data processing. To solve these problems successfully, the data has to be stored, manipulated, and managed effectively. This topic will discuss at a high level the different kinds of tasks you, as a developer, might have to undertake. Later in this course, other modules will expand on these tasks in more detail.

Storing and managing data

The primary SQL Server object for storing and retrieving data is the table. Creating, deleting, and altering tables are some of the most important tasks you will perform. After a table is created, you can insert data into a table, amend data in a table, and move data between tables.

With SQL Server views, you can create a single logical view across multiple tables, filtering data so that only relevant information is returned.

Indexes can also be added to tables to ensure good performance when querying the data. As the volume of data becomes greater, ensuring good performance becomes important.

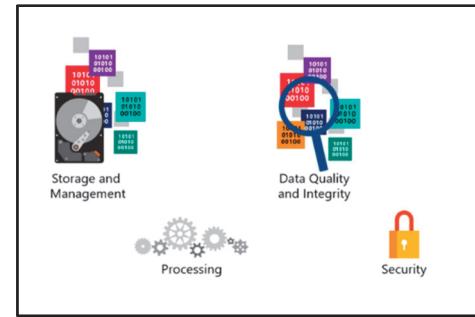
Processing data programmatically

SQL Server encapsulates business logic through the use of stored procedures and functions. Rather than build business logic into multiple applications, client applications can call stored procedures and functions to perform data operations. This centralizes business logic, and makes it easier to maintain.

SSIS supports more complex data processing in the form of extraction and transformation.

Enforcing and inspecting data quality

Data quality is maintained by placing constraints on columns in a table. For example, you can specify a data type to restrict the type of data that can be stored. This constrains the column to only holding, for example, integers, date and time, or character data types. Columns can be further constrained by specifying the length of the data type, or whether or not it can be left empty (null).



Primary keys and unique keys ensure a value is unique amongst other rows in the table. You can also link tables together by creating foreign keys.

If a database is poorly designed, without using any of these data quality constraints, a developer might have to inspect and resolve data issues—removing duplicate data or performing some kind of data cleansing.

Securing data

This can include:

- Restricting access to the data to certain groups or specific individuals.
- Protecting data by using a backup routine to ensure nothing is lost in the case of a hardware failure.
- Encrypting data to reduce the possibility of unauthorized access to sensitive data.

These tasks are usually performed by a database administrator. However, developers often provide guidance about who needs access to what data; the business requirements around availability and scheduling of backups; and what form of encryption is required.

Development Tools for SQL Server

SQL Server provides a range of tools for developers to complete the previously described tasks. In some circumstances, more than one tool might be suitable.

SQL Server Management Studio

SSMS is the primary tool that Microsoft provides for interacting with SQL Server services. It is an integrated environment that exists in the Visual Studio platform shell. SSMS shares many common features with Visual Studio.

You use SSMS to execute queries and return results, but it can also help you to analyze queries. It offers rich editors for a variety of document types, including SQL files and xml files. When you are working with SQL files, SSMS provides IntelliSense® to help you write queries.

You can perform most SQL Server relational database management tasks by using the Transact-SQL language, but many developers prefer the graphical administration tools because they are typically easier to use. SSMS provides graphical interfaces for configuring databases and servers.

SSMS can connect to a variety of SQL Server services, including the Database Engine, Analysis Services, Integration Services, and Reporting Services. SSMS uses the Visual Studio environment and will be familiar to Visual Studio developers.

SQL Server Data Tools

SSDT brings SQL Server functionality into Visual Studio. With SSDT, Visual Studio you can develop both on-premises, and cloud-based applications using SQL Server components. You can work with .NET Framework code and database-specific code, such as Transact-SQL, in the same environment. If you want to change the database design, you do not have to leave Visual Studio and open SSMS; you can work with the schema within SSDT.

- SQL Server Management Studio
- SQL Server Data Tools
- Visual Studio Code (VS Code)
- SQL Operations Studio
- SQL Server Profiler
- Database Engine Tuning Advisor
- Data Quality Services client

Visual Studio Code

Visual Studio Code (VS Code) is a free, open source code editor for Windows, Linux, and macOS. VS Code uses an extension framework to add functionality to the base editor. To connect to SQL Server from VS Code, you must install the **mssql** extension, that gives you the capability to connect to SQL Server and execute Transact-SQL commands. You can download VS Code from Microsoft.

SQL Operations Studio

SQL Operations Studio is a free, lightweight administration tool for SQL Server than runs on Windows, Linux, and macOS. SQL Operations Studio offers many of the same features as SQL Server Management Studio, and includes new features such as customizable dashboards that you can use to get an overview of server performance. At the time of writing, SQL Operations Studio is in public preview, and features are subject to change. You can download VS Code from Microsoft.

SQL Server Profiler

SQL Server Profiler is a graphical user interface tool that is used to view the output of a SQL Trace. You use SQL Server Profiler to monitor the performance of the Database Engine or Analysis Services by capturing traces and saving them to a file or table. You can use Profiler to step through problem queries to investigate the causes.

You can also use the saved trace to replicate the problem on a test server, making it easier to diagnose the problem.

Profiler also supports auditing an instance of SQL Server. Audits record security-related actions so they can be reviewed later.

Database Engine Tuning Advisor

The Database Engine Tuning Advisor analyzes databases and makes recommendations that you can use to optimize performance. You can use it to select and create an optimal set of indexes, indexed views, or table partitions. Common usage includes the following tasks:

- Identify and troubleshoot the performance of a problem query.
- Tune a large set of queries across one or more databases.
- Perform an exploratory what-if analysis of potential physical design changes.

Data Quality Services (DQS) Client

With the DQS client application, you can complete various data quality operations, including creating knowledge bases, creating and running data quality projects, and performing administrative tasks. After completing these operations, you can perform a number of data quality tasks, including correction, enrichment, standardization, and deduplication of data.

Demonstration: Using SSMS and SSDT

In this demonstration, you will see how to:

- Use SSMS to connect to an on-premises instance of SQL Server.
- Run a Transact-SQL script.
- Open an SSMS project.
- Connect to servers and databases.
- Use SSDT to run a Transact-SQL script.

Demonstration Steps

Use SSMS to Connect to an On-premises Instance of SQL Server

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running and then log on to **20762C-MIA-SQL** as **AdventureWorks\Student** with the password **Pa55w.rd**.
2. Navigate to **D:\Demofiles\Mod01**, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**.
4. On the taskbar, click **Microsoft SQL Server Management Studio**.
5. In the **Connect to Server** dialog box, ensure that **Server type** is set to **Database Engine**.
6. In the **Server name** text box, type **MIA-SQL**.
7. In the **Authentication** drop-down list, select **Windows Authentication**, and then click **Connect**.

Run a Transact-SQL Script

1. In Object Explorer, expand **Databases**, expand **AdventureWorks**, and then review the database objects.
2. In Object Explorer, right-click **AdventureWorks**, and then click **New Query**.
3. Type the query shown in the snippet below:

```
SELECT * FROM Production.Product ORDER BY ProductID;
```

Note the use of IntelliSense while you are typing this query, and then on the toolbar, click **Execute**. Note how the results can be returned.

4. On the **File** menu, click **Save SQLQuery1.sql As**.
5. In the **Save File As** dialog box, navigate to **D:\Demofiles\Mod01**, and then click **Save**. Note that this saves the query to a file.
6. On the **Results** tab, right-click the cell for **ProductID 1** (first row and first cell), and then click **Save Results As**.
7. In the **Save Grid Results** dialog box, navigate to the **D:\Demofiles\Mod01** folder.
8. In the **File name** box, type **Demonstration2AResults**, and then click **Save**. Note that this saves the query results to a file.
9. On the **Query** menu, click **Display Estimated Execution Plan**. Note that SSMS can do more than just execute queries.
10. On the **Tools** menu, click **Options**.

11. In the **Options** dialog box, expand **Query Results**, expand **SQL Server**, and then click **General**. Review the available configuration options, and then click **Cancel**.

12. On the **File** menu, click **Close**.

Open a SQL Server Management Studio Project

1. On the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, open the **D:\Demofiles\Mod01\Demo01.ssmssln** project.
3. In Solution Explorer, note the contents of Solution Explorer, and that by using a project or solution you can save the state of the IDE. This means that any open connections, query windows, or Solution Explorer panes will reopen in the state they were saved in.
4. On the **File** menu, click **Close Solution**.

Connect to Servers and Databases

1. In Object Explorer, click **Connect** and note the other SQL Server components to which connections can be made.
2. On the **File** menu, point to **New**, and then click **Database Engine Query** to open a new connection.
3. In the **Connect to Database Engine** dialog box, in the **Server name** box, type **MIA-SQL**.
4. In the **Authentication** drop-down list, select **Windows Authentication**, and then click **Connect**.
5. In the **Available Databases** drop-down list on the toolbar, click **tempdb**. Note that this changes the database against which the query is executed.
6. Right-click in the query window, point to **Connection**, and then click **Change Connection**. This will reconnect the query to another instance of SQL Server.
7. In the **Connect to Database Engine** dialog box, click **Cancel**.
8. Close SSMS.

Use SSDT to Run a Transact-SQL Script

1. On the taskbar, click **Visual Studio 2015**.
2. On the **Tools** menu, click **Connect to Database**.
3. In the **Choose Data Source** dialog box, in the **Data source** list, click **Microsoft SQL Server**, and then click **Continue**.
4. In the **Add Connection** dialog box, in the **Server name** box, type **MIA-SQL**.
5. In the **Select or enter a database name** drop-down list, click **AdventureWorks**, and then click **OK**.
6. In Server Explorer, expand **Data Connections**.
7. Right-click **mia-sql.AdventureWorks.dbo**, and then click **New Query**.
8. In the **SQLQuery1.sql** pane type:

```
SELECT * FROM Production.Product ORDER BY ProductID;
```

9. On the toolbar, click **Execute**.
10. Note that you can view results, just as you can in SSMS.
11. Close Visual Studio 2015 without saving any changes.

Check Your Knowledge

Question	
Which one of the following tools can you use to create and deploy SSIS packages?	
Select the correct answer.	
<input type="checkbox"/>	SQL Server Management Studio.
<input type="checkbox"/>	SQL Server Profiler.
<input type="checkbox"/>	Database Engine Tuning Advisor.
<input type="checkbox"/>	SQL Server Data Tools.
<input type="checkbox"/>	SQL Server Configuration Manager.

Module Review and Takeaways

Review Question(s)

Question: Which IDE do you think you will use to develop on SQL Server, SSMS or SSDT?

Module 2

Designing and Implementing Tables

Contents:

Module Overview	2-1
Lesson 1: Designing Tables	2-2
Lesson 2: Data Types	2-12
Lesson 3: Working with Schemas	2-26
Lesson 4: Creating and Altering Tables	2-31
Lab: Designing and Implementing Tables	2-38
Module Review and Takeaways	2-41

Module Overview

In a relational database management system (RDBMS), user and system data is stored in tables. Each table consists of a set of rows that describe entities and a set of columns that hold the attributes of an entity. For example, a Customer table might have columns such as **CustomerName** and **CreditLimit**, and a row for each customer. In Microsoft SQL Server® data management software tables are contained within schemas that are very similar in concept to folders that contain files in the operating system. Designing tables is one of the most important tasks that a database developer undertakes, because incorrect table design leads to the inability to query the data efficiently.

After an appropriate design has been created, it is important to know how to correctly implement the design.

Objectives

After completing this module, you will be able to:

- Design tables using normalization, primary and foreign keys.
- Work with identity columns.
- Understand built-in and user data types.
- Use schemas in your database designs to organize data, and manage object security.
- Work with computed columns and temporary tables.

Lesson 1

Designing Tables

The most important aspect of designing tables involves determining what data each column will hold. All organizational data is held within database tables, so it is critical to store the data with an appropriate structure.

The best practices for table and column design are often represented by a set of rules that are known as “normalization” rules. In this lesson, you will learn the most important aspects of normalized table design, along with the appropriate use of primary and foreign keys. In addition, you will learn to work with the system tables that are supplied when SQL Server is installed.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe what a table is.
- Normalize data.
- Describe common normal forms.
- Explain the role of primary keys.
- Explain the role of foreign keys.
- Work with system tables.
- Design tables for concurrency.
- Use identity columns and sequences.

What Is a Table?

Relational databases store data about entities in tables, and tables are defined by columns and rows. Rows represent entities, and columns define the attributes of the entities. By default, the rows of a table have no predefined order, and can be used as a security boundary.

Tables

In the terminology of formal relational database management systems, tables are referred to as “relations.”

Tables store data about entities such as customers, suppliers, orders, products, and sales. Each row of a table represents the details of a single entity, such as a single customer, supplier, order, product, or sale.

Columns define the information that is being held about each entity. For example, a **Product** table might have columns such as **ProductID**, **Size**, **Name**, and **UnitWeight**. Each of these columns is defined by using a specific data type. For example, the **UnitWeight** column of a product might be allocated a **decimal** (18,3) data type.

- Relational databases store data in tables (relations)
 - Defined by a collection of columns (identified by name)
 - Contain zero or more rows
- Tables typically represent a type of object or entity
 - Employees, purchase orders, customers, and sales orders are examples of entities
 - Consistent naming convention for tables is important
- Tables are a security boundary
 - Each row usually represents a single instance of the object or entity
 - One employee, or one purchase order, for example
 - Rows of tables have no order

Naming Conventions

There is strong disagreement within the industry over naming conventions for tables. The use of prefixes (such as `tblCustomer` or `tblProduct`) is widely discouraged. Prefixes were commonly used in higher level programming languages before the advent of strong data typing—that is, the use of strict data types rather than generic data types—but are now rare. The main reason for this is that names should represent the entities, not how they are stored. For example, during a maintenance operation, it might become necessary to replace a table with a view, or vice versa. This could lead to views named `tblProduct` or `tblCustomer` when trying to avoid breaking existing code.

Another area of strong disagreement relates to whether table names should be singular or plural. For example, should a table that holds the details of a customer be called `Customer` or `Customers`? Proponents of plural naming argue that the table holds the details of many customers; proponents of singular naming say that it is common to expose these tables via object models in higher level languages, and that the use of plural names complicates this process. Although we might have a `Customers` table, in a high level language, we are likely to have a `Customer` object. SQL Server system tables and views have plural names.

The argument is not likely to be resolved either way and is not a problem that is specific to the SQL language. For example, an array of customers in a higher level language could sensibly be called “`Customers`,” yet referring to a single customer via “`Customers[49]`” seems awkward. The most important aspect of naming conventions is that you should adopt one that you can work with and apply consistently.

Security

You can use tables as security boundaries because users can be assigned permissions at the table level. However, note that SQL Server supports the assignment of permissions at the column level, in addition to the table level; row-level security is available for tables in SQL Server. Row, column, and table security can also be implemented by using a combination of views, stored procedures, and/or triggers.

Row Order

Tables are containers for rows but, by default, they do not define any order for the rows that they contain. When users select rows from a table, they should only specify the order that the rows should be returned in if the output order matters. SQL Server might have to expend additional sorting effort to return rows in a given order, and it is important that this effort is only made when necessary.

Normalizing Data

Normalization is a systematic process that you can use to improve the design of databases.

Normalization

Edgar F. Codd (1923–2003) was an English computer scientist who is widely regarded as having invented the relational model. This model underpins the development of relational database management systems. Codd introduced the concept of normalization and helped it evolve over many years, through a series of “normal forms.”

- Normalization is a process
 - Ensures that database structures are appropriate
 - Ensures that poor design characteristics are avoided
- Edgar F. Codd invented the relational model
 - Introduced the concept of normalization
 - Referred to the degrees of normalization as forms
- Database designs should initially be normalized
 - Denormalization might be applied later to improve performance or to make analysis of data more straightforward

Codd introduced first normal form in 1970, followed by second normal form, and then third normal form in 1971. Since that time, higher forms of normalization have been introduced by theorists, but most database designs today are considered to be "normalized" if they are in third normal form.

Intentional Denormalization

Not all databases should be normalized. It is common to intentionally denormalize databases for performance reasons or for ease of end-user analysis.

For example, dimensional models that are widely used in data warehouses (such as the data warehouses that are commonly used with SQL Server Analysis Services) are intentionally designed not to be normalized.

Tables might also be denormalized to avoid the need for time-consuming calculations or to minimize physical database design constraints, such as locking.

Common Normalization Forms

In general, normalizing a database design leads to an improved design. You can avoid most common table design errors in database systems by applying normalization rules.

Normalization

You should use normalization to:

- Free the database of modification anomalies.
- Minimize redesign when the structure of the database needs to be changed.
- Ensure that the data model is intuitive to users.
- Avoid any bias toward particular forms of querying.

Although there is disagreement on the interpretation of these rules, there is general agreement on most common symptoms of violating the rules.

First Normal Form

To adhere to the first normal form, you must eliminate repeating groups in individual tables. To do this, you should create a separate table for each set of related data, and identify each set of related data by using a primary key.

For example, a **Product** table should not include columns such as **Supplier1**, **Supplier2**, and **Supplier3**. Column values should not include repeating groups. For example, a column should not contain a comma-separated list of suppliers.

Duplicate rows should not exist in tables. You can use unique keys to avoid having duplicate rows. A candidate key is a column or set of columns that you can use to uniquely identify a row in a table. An alternate interpretation of first normal form rules would disallow the use of nullable columns.

Second Normal Form

To adhere to second normal form, you must create separate tables for sets of values that apply to multiple records, and relate these tables by using a foreign key.

- | |
|---|
| <ul style="list-style-type: none">• First Normal Form<ul style="list-style-type: none">• Eliminate repeating groups in individual tables• Create a separate table for each set of related data• Identify each set of related data by using a primary key• Second Normal Form<ul style="list-style-type: none">• Non-key columns should not be dependent on only part of a primary key• These columns should be in a separate table and related by using a foreign key• Third Normal Form<ul style="list-style-type: none">• Eliminate fields that do not depend on the key |
|---|

For example, a second normal form error would be to hold the details of products that a supplier provides in the same table as the details of the supplier's credit history. You should store these values in a separate table.

Third Normal Form

To adhere to third normal form, eliminate fields that do not depend on the key.

Imagine a **Sales** table that has **OrderNumber**, **ProductID**, **ProductName**, **SalesAmount**, and **SalesDate** columns. This table is not in third normal form. A candidate key for the table might be the **OrderNumber** column. However, the **ProductName** column only depends on the **ProductID** column and not on the candidate key, so the **Sales** table should be separated from a **Products** table, and probably linked to it by **ProductID**.

Formal database terminology is precise, but can be hard to follow when it is first encountered. In the next demonstration, you will see examples of common normalization errors.

Demonstration: Working with Normalization

In this demonstration, you will see how to alter a table to conform to third normal form.

Demonstration Steps

1. Ensure that the **MT17B-WS2016-NAT**, **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **AdventureWorks\Student** with the password **Pa55w.rd**.
2. In File Explorer, navigate 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. On the taskbar, click **Microsoft SQL Server Management Studio**.
5. In the **Connect to Server** dialog box, connect to **MIA-SQL**, using **Windows Authentication**.
6. On the **File** menu, point to **Open**, click **Project/Solution**.
7. In the **Open Project** dialog box, navigate to the **D:\Demofiles\Mod02** folder, click **Demo.ssmssln**, and then click **Open**.
8. In Solution Explorer, expand **Queries**, and then double-click **1 - Normalization.sql**.
9. Select the code under the **Step 1: Set AdventureWorks as the current database** comment, and then click **Execute**.
10. Select the code under the **Step 2: Create a table for denormalizing** comment, and then click **Execute**.
11. Select the code under the **Step 3: Alter the table to conform to third normal form** comment, and then click **Execute**.
12. Select the code under the **Step 4: Drop and recreate the ProductList table** comment, and then click **Execute**.
13. Select the code under the **Step 5: Populate the ProductList table** comment, and then click **Execute**.
14. Close SQL Server Management Studio without saving any changes.

Primary Keys

A primary key is a form of constraint that uniquely identifies each row within a table. A candidate key is a column or set of columns that you could use to identify a row uniquely—it is a candidate to be chosen for the primary key. A primary key must be unique and cannot be NULL.

Consider a table that holds an **EmployeeID** column and a **NationalIDNumber** column, along with the employee's name and personal details.

The **EmployeeID** and **NationalIDNumber**

columns are both candidate keys. In this case, the **EmployeeID** column might be the primary key,

but either candidate key could be used. You will see later that some data types will lead to better performing systems when they are used as primary keys, but logically any candidate key can be nominated to be the primary key.

It might be necessary to combine multiple columns into a key before the key can be used to uniquely identify a row. In formal database terminology, no candidate key is more important than any other candidate key. However, when tables are correctly normalized, they will usually have only a single candidate key that could be used as a primary key. Ideally, keys that are used as primary keys should not change over time.

Natural vs. Surrogate Keys

Natural keys are formed from data within the table. A surrogate key is another form of key that you can use as a unique identifier within a table, but it is not derived from "real" data.

For example, a **Customer** table might have a **CustomerID** or **CustomerCode** column that contains numeric, GUID, or alphanumeric codes. The surrogate key would not be related to the other attributes of a customer.

The use of surrogate keys is another subject that can lead to strong debate between database professionals.

- The primary key uniquely identifies each row within a table
- Candidate key could be used to uniquely identify a row
 - Must be unique and cannot be NULL (unknown)
 - Can involve multiple columns
 - Should not change
 - Primary key is one candidate key
 - Most tables will only have a single candidate key
- Debate surrounding natural vs. surrogate keys
 - Natural key: formed from data related to the entity
 - Surrogate key: usually codes or numbers

Foreign Keys

A foreign key is a key in one table that matches, or references, a unique key in another table. Foreign keys are used to create a relationship between tables. A foreign key is a constraint because it limits the data that can be held in the field to a value that matches a field in the related table.

To create a foreign key, you should create a field of the same data type as the unique or primary key of another table. This will then be related to the table containing the unique or primary key in the table definition. You can create a foreign key using either CREATE TABLE or ALTER TABLE.

- Foreign keys are references between tables:
 - Foreign key in one table holds the primary key from another table
 - Self-references are permitted
 - Rows that do not exist in the referenced table cannot be inserted in a referencing table
 - Rows cannot be deleted or updated without cascading options
 - Multiple foreign keys can exist in one table

For example, a **CustomerOrders** table might include a **CustomerID** column. A foreign key reference is used to ensure that any **CustomerID** value that is entered in the **CustomerOrders** table does in fact exist in the **Customers** table.

In SQL Server, the reference is only checked if the column that holds the foreign key value is not NULL.

Self-Referencing Tables

A table can hold a foreign key reference to itself. For example, an **Employees** table might contain a **ManagerID** column. An employee's manager is also an employee; therefore, a foreign key reference can be made from the **ManagerID** column of the **Employees** table to the **EmployeeID** column in the same table.

Reference Checking

SQL Server cannot update or delete referenced keys unless you enable options that cascade the changes to related tables. For example, you cannot change the ID for a customer when there are orders in a **CustomerOrders** table that reference the customer's ID.

Tables might also include multiple foreign key references. For example, an **Orders** table might have foreign keys that refer to both a **Customers** table and a **Products** table.

Terminology

Foreign keys are used to "enforce referential integrity." Foreign keys are a form of constraint and will be covered in more detail in a later module.

The ANSI SQL 2003 definition refers to self-referencing tables as having "recursive foreign keys."

Working with System Tables

System tables are the tables provided with the SQL Server Database Engine. You should not directly modify system tables, or query them directly. Catalog views are used to get information from system tables.

System Tables in Earlier Versions

If you have worked with SQL Server 2000 or earlier versions, you might be expecting databases to contain a large number of system tables.

Users sometimes modified these system tables, and this caused issues when applying service packs or updates. More seriously, this could lead to unexpected behavior or failures if the data was not changed correctly. Also, users often took dependencies on the format of these system tables. That made it difficult for new versions of SQL Server to improve the table designs without breaking existing applications. As an example, when it was necessary to expand the **syslogins** table, a new **sysxlogins** table was added instead of changing the existing table.

In SQL Server 2005, system tables were hidden and replaced by a set of system views that show the contents of the system tables. These views are permission-based and display data to a user only if the user has appropriate permission.

- SQL Server provides a set of system tables
 - Should not be directly modified or queried
- In SQL Server 2005, most system tables were replaced by a set of permission-based system views
- Some system tables in the **msdb** database are still useful
 - **dbo.backupset**
 - **dbo.restorehistory**
 - **dbo.sysjobhistory**

System Tables in the msdb Database

SQL Server Agent uses the **msdb** database, primarily for organizing scheduled background tasks that are known as "jobs." A large number of system tables are still present in the **msdb** database. Again, while it is acceptable to query these tables, they should not be directly modified. Unless the table is documented, no dependency on its format should be taken when designing applications.

Designing for Concurrency

SQL Server uses locking to manage transaction concurrency. When a transaction modifies data, it acquires an appropriate lock to protect the data. The lock resources refer to the resource being locked by the transaction; for example, row, page, or table. All locks held by transactions are released when the transaction commits.

SQL Server minimizes the locking cost by acquiring locks at a level appropriate for a task. Locking at lower granularity levels, such as rows, increases concurrency but has an overhead of maintaining a large number of locks used to lock many rows. Locking at higher granularity levels, such as tables, decreases concurrency because the entire table is inaccessible to other transactions. However, the overhead is less because fewer locks need to be maintained.

- SQL Server uses locking to manage concurrency:
 - Locks can be made at the row, page, or table level
 - Row locking increases concurrency, but there is an overhead of maintaining many locks
- OLTP databases should be normalized to increase concurrency:
 - Transactions should be small and fast
 - Smaller tables (fewer columns) less likely to cause locking issues
- Data warehouse tables should be denormalized—no modifications to create locking problems

Designing for an OLTP System

The way in which your database is used will determine the best design for concurrency. You may need to monitor your database over time to determine if the design is sufficient, and make alterations if locking becomes a frequent problem. Your goal is to ensure transactions are as small and as fast as possible, and less likely to block other transactions.

The higher the number of users in your database, the more locking you will have, because they are more likely to simultaneously access the same row, table, and pages. The more locking you have, the lower the performance of the system, because one user must wait for another user to finish their transaction, and the application may temporarily freeze. You may also find that there are certain times of the day when things slow down, such as afternoons, when staff return to the office after lunch. One option is to change the transaction isolation level; however, this creates other potential problems, with logical errors in your data. The better solution is to use normalization to separate the data as much as possible. While this creates extra joins in your queries, it does help with concurrency.

If you have a table with 25 columns, and three users attempt to modify three different columns in the same row, SQL Server takes a row lock for the first user, and the other two must wait for the first user to complete. The third user must wait for the first two to complete. By splitting the table into three tables, and separating the columns that are likely to be modified, each user will be modifying a different table, without blocking the other users.

Designing for a Data Warehouse

In a data warehouse, you won't have lots of users making modifications, so locking won't be an issue. Users will only read data from the data warehouse, so it is safe to denormalize the data, because one table does not need to describe one entity.

Implementing Surrogate Keys

If a table has no natural key, you must use a surrogate key. A surrogate key is not part of the original data, but is added to uniquely identify the row. **IDENTITY** and **SEQUENCE** both produce unique surrogate keys, but are used in different situations.

IDENTITY

The **IDENTITY** property creates a unique value, based on the seed and the increment values provided. These are normally sequential values.

As you learned in a previous lesson, a primary key column uniquely identifies each row. When designing tables, you will find that some entities have a natural primary key, such as **CustomerCode**, whereas **Categories** might not. Whether you decide against using a natural primary key, or use a surrogate key because there is no natural key, the **IDENTITY** property will create a sequence of unique numbers.

The **IDENTITY** property is used in a **CREATE TABLE** or **ALTER TABLE** statement, and requires two values: the starting seed, and the increment.

In the following example, the first row in the **Categories** table will have a **CategoryID** value of **1**. The next row will have a value of **2**:

In the following example, the first row in the **Categories** table will have a **CategoryID** value of **1**. The next row will have a value of **2**:

IDENTITY Property

```
CREATE TABLE Categories
(
    CategoryID int IDENTITY(1,1),
    Category varchar(25) NOT NULL
);
```

The data type of the column is **int**, so the starting seed can be any value that the integer stores, but it is common practice to start at **1**, and increment by **1**.

When inserting a record into a table with an identity column, if a transaction fails and is rolled back, then the value that the row would have been assigned as the identity is not used for the next successful insert.

IDENTITY_INSERT

You can insert an explicit value into an identity column using **SET IDENTITY_INSERT ON**. The insert value cannot be a value that has already been used; however, you can use a value from a failed transaction that was not used, and keep the number sequence intact. You can also use the value from a row that has been deleted. There can only be one **IDENTITY_INSERT** per session, so be sure to include **SET IDENTITY_INSERT OFF** after the insert statement has completed.

```
SET IDENTITY_INSERT Categories ON
INSERT INTO Categories (CategoryID, Category)
VALUES (5, 'Cat Food')
SET IDENTITY_INSERT Categories OFF
```

- Use **IDENTITY** with **CREATE** or **ALTER TABLE** to create a unique, sequentially numbered column
- **SET IDENTITY_INSERT ON** allows explicit values to be inserted into a column with **IDENTITY**
- **@@IDENTITY** returns the last value created by the **IDENTITY** column across all sessions
- **SCOPE_IDENTITY()** returns the last value created by the **IDENTITY** column for the current session
- **SEQUENCE** creates a numbered list that can be used by all tables in the database
 - The value increments each time the next value is requested
 - Can be restarted
 - **MINVALUE** and **MAXVALUE** set boundaries

Deleting Rows from a Table with an Identity Column

After deleting all rows from a table with an identity column, the seed value of the next inserted row will be the next number in sequence. If the last value was 237, then the next ID will be 238. There are two ways of reseeding the table:

- Use **TRUNCATE TABLE** instead of **DELETE FROM**, though keep in mind that truncate is only minimally logged in the transaction log:

```
TRUNCATE TABLE CATEGORIES;
```

- Run the DBCC CHECKIDENT command to reseed the table. The reseed value should be **0** to set the first row back to **1**:

```
DBCC CHECKIDENT('Categories', RESEED, 0)
```

@@IDENTITY vs. SCOPE_IDENTITY()

@@IDENTITY and **SCOPE_IDENTITY()** are very similar in function, but with a subtle difference. Both return the identity value of the last inserted record. However, **@@IDENTITY** returns the last insert, regardless of session, whereas **SCOPE_IDENTITY()** will return the value from the current session. If you insert a row and need the identity value, use **SCOPE_IDENTITY()**—if you use **@@IDENTITY** after an insert, and another session also inserts a new row, you might pick up that value, rather than your row value.

```
INSERT INTO Categories (Category)
VALUES ('Dog Food')
SELECT SCOPE_IDENTITY() AS CategoryID;

CategoryID
-----
6
```

SEQUENCE

The SEQUENCE object performs a similar function to IDENTITY, but has a lot more flexibility. You create a SEQUENCE object at the database level, and values can be used by multiple tables. Whereas IDENTITY creates a new value with each row insert, SEQUENCE returns a value when requested. This value does not need to be inserted into a table:

```
CREATE SEQUENCE CategoryID AS int
START WITH 1
INCREMENT BY 1;
```

SEQUENCE is useful when you want control over the values you are inserting. In the above example, the Categories table uses IDENTITY to generate the **CategoryID** values. Consider if there were two tables in your database, one for **MainCategories**, and another for **SubCategories**, but you wanted the **CategoryID** to be unique across both tables—you could use **SEQUENCE**.

NEXT VALUE FOR generates the next value.

NEXT VALUE FOR

```
INSERT INTO MainCategories (CategoryID, MainCategory)
VALUES
(NEXT VALUE FOR CategoryID, 'Food'),
(NEXT VALUE FOR CategoryID, 'Drink');
GO

INSERT INTO SubCategories (CategoryID, SubCategory)
VALUES
(NEXT VALUE FOR CategoryID, 'Cat Food'),
(NEXT VALUE FOR CategoryID, 'Wine');
```

Assuming both tables were empty, the results would be as follows:

Assuming both tables were empty, the results would be as follows:

Results of inserting rows using the SEQUENCE object

CategoryID	MainCategory
1	Food
2	Drink

CategoryID	SubCategory
3	Cat Food
4	Wine

Restart the Sequence

You can restart your sequence at any time, though consider the consequences if you have a primary key on the columns you are using the sequence values for, as values must be unique:

```
ALTER SEQUENCE CategoryID
RESTART WITH 1;
```

MINVALUE and MAXVALUE

The sequence can be limited by using the **MINVALUE** and **MAXVALUE** properties—when you reach the **MAXVALUE** limit, you will receive an error.

The sequence can be limited by using the **MINVALUE** and **MAXVALUE** properties—when you reach the **MAXVALUE** limit, you will receive an error.

MINVALUE and MAXVALUE

```
ALTER SEQUENCE CategoryID
MINVALUE 1
MAXVALUE 2000;
```



Note: If you want to know the next value in the sequence, you can run the code: **SELECT NEXT VALUE FOR CategoryID**. However, each time this runs, the sequence value will increment, even if you don't use this value in an insert statement.

Question: Would it be reasonable to have columns called, for example, **AddressLine1**, **AddressLine2**, and **AddressLine3** in a normalized design?

Lesson 2

Data Types

The most basic types of data that get stored in database systems are numbers, dates, and strings. There is a range of data types that can be used for each of these. In this lesson, you will see the Microsoft-supplied data types that you can use for numeric and date-related data. You will also see what NULL means and how to work with it. In the next lesson, you will see how to work with string data types.

Lesson Objectives

After completing this lesson, you will be able to:

- Understand the purpose of data types.
- Use exact numeric data types.
- Use approximate numeric data types.
- Use date and time data types.
- Work with unique identifiers.
- Understand when to use NULL and NOT NULL.
- Create alias data types.
- Convert data between data types.
- Work with international character data.

Introduction to Data Types

Data types determine what can be stored in locations within SQL Server, such as columns, variables, and parameters. For example, a **tinyint** column can only store whole numbers from 0 to 255. Data types also determine the types of values that can be returned from expressions.

Constraining Values

Data types are a form of constraint that is placed on the values that can be stored in a location. For example, if you choose a numeric data type, you will not be able to store text.

In addition to constraining the types of values that can be stored, data types also constrain the range of values that can be stored. For example, if you choose a **smallint** data type, you can only store values between -32,768 and +32,767.

Query Optimization

When SQL Server identifies that the value in a column is an integer, it might be able to generate an entirely different and more efficient query plan to one where it identifies that the location is holding text values.

The data type also determines which sorts of operations are permitted on that data and how those operations work.

- Data types determine what can be stored
 - Constrain the type of data that an object can hold and the permitted operations
 - Provide limits on the range of values
- Data types apply to database columns, variables, expressions, and parameters
- Critical to choose appropriate data types
 - Assist with query optimization
 - Provide a level of self-documentation
- Three basic sets of data types
 - System data types
 - Alias data types
 - User-defined data types

Self-Documenting Nature

Choosing an appropriate data type provides a level of self-documentation. If all values were stored in a string value (which could potentially represent any type of value) or XML data types, you would probably need to store documentation about what sort of values can be stored in the string locations.

Data Types

There are three basic sets of data types:

- **System data types.** SQL Server provides a large number of built-in (or intrinsic) data types. Examples of these include **integer**, **varchar**, and **date**.
- **Alias data types.** Users can also define data types that provide alternate names for the system data types and, potentially, further constrain them. These are known as alias data types. For example, you could use an alias data type to define the name **PhoneNumber** as being equivalent to **nvarchar(16)**. Alias data types can help to provide consistency of data type usage across applications and databases.
- **User-defined data types.** By using managed code via SQL Server integration with the common language runtime (CLR), you can create entirely new data types. There are two categories of these CLR types. One category is system CLR data types, such as the **geometry** and **geography** spatial data types. The other is user-defined CLR data types, which enable users to create their own data types.

Exact Numeric Data Types

Numeric data types can be exact or approximate. Exact data types are the most common data type that is used in business applications.

Integer Data Types

SQL Server offers a choice of integer data types that are used for storing whole numbers, based upon the size of the storage location for each:

- **tinyint** is stored in a single byte (that is, 8 bits) and can be used to store the values 0 to 255. Note that, unlike the other integer data types, **tinyint** cannot store any negative values.
- **smallint** is stored in 2 bytes (that is, 16 bits) and stores values from -32,768 to 32,767.
- **int** is stored in 4 bytes (that is, 32 bits) and stores values from -2,147,483,648 to 2,147,483,647. It is a very commonly used data type. SQL Server uses the full word "integer" as a synonym for "int."
- **bigint** is stored in 8 bytes (that is, 64 bits) and stores very large integer values. Although it is easy to refer to a 64-bit value, it is hard to comprehend how large these values are. If you placed a value of zero in a 64-bit integer location, and executed a loop to add one to the value, on most common servers currently available, you would not reach the maximum value for many months.

- Numeric types: range, precision, and accuracy
 - tinyint: 8 bits (0 to 255)
 - smallint: 16-bit integer (-32,768 to 32,767)
 - int: 32-bit integer (-2,147,483,648 to 2,147,483,647)
 - bigint: 64-bit integer (-2^63 to 2^63 - 1)
 - decimal: fixed precision and scale (-10^38+1 to 10^38-1)
 - numeric: functionally equivalent to decimal
 - smallmoney: fixed scale of four decimal places in 32 bits—avoid
 - money: fixed scale of four decimal places in 64 bits—avoid
 - bit values of 1, 0, or NULL

Exact Fractional Data Types

SQL Server provides a range of data types for storing exact numeric values that include decimal places:

- **decimal** is an ANSI-compatible data type you use to specify the number of digits of precision and the number of decimal places (referred to as the scale). A **decimal(12,5)** location can store up to 12 digits with up to five digits after the decimal point. You should use the **decimal** data type for

monetary or currency values in most systems, and any exact fractional values, such as sales quantities (where part quantities can be sold) or weights.

- **numeric** is a data type that is functionally equivalent to **decimal**.
- **money** and **smallmoney** are data types that are specific to SQL Server and have been present since the early days of the platform. They were used to store currency values with a fixed precision of four decimal places.



Note: Four is often the wrong number of decimal places for many monetary applications, and the **money** and **smallmoney** data types are not standard data types. In general, use **decimal** for monetary values.

bit Data Type

bit is a data type that is stored in a single bit. The storage of the **bit** data type is optimized. If there are eight or fewer **bit** columns in a table, they are stored in a single byte. **bit** values are commonly used to store the equivalent of Boolean values in higher level languages.

Note that there is no literal string format for **bit** values in SQL Server. The string values TRUE and FALSE can be converted to **bit** values, as can the integer values 1 and 0. TRUE is converted to 1 and FALSE is converted to 0.

Higher level programming languages differ on how they store true values in Boolean columns. Some languages store true values as 1; others store true values as -1. To avoid any chance of mismatch, in general, when working with bits in applications, test for false values by using the following code:

```
IF (@InputValue = 0)
```

Test for positive values by using the following code:

```
IF (@InputValue <> 0)
```

This is preferable to testing for a value being equal to 1 because it will provide more reliable code.

bit, along with other data types, is also nullable, which can be a surprise to new users. That means that a **bit** location can be in three states: NULL, 0, or 1. (Nullability is discussed in more detail later in this module.)

To learn more about data types, see Microsoft developer documentation:



Data Types (Transact-SQL)

<http://go.microsoft.com/fwlink/?LinkId=233787>

Approximate Numeric Data Types

SQL Server provides two approximate numeric data types. They are used more commonly in scientific applications than in business applications. A common design error is to use the **float** or **real** data types for storing business values such as monetary values.

- Two approximate numeric types are supported:
 - **float** is from **float(1)** to **float(53)**
 - **float** defaults to **float(53)**
 - **real** is fixed 4-byte storage
- **float** and **real** are not regularly used in business applications because they are not precise

Approximate Numeric Values

The **real** data type is a 4-byte (that is, 32-bit) numeric value that is encoded by using ISO standard floating-point encoding.

The **float** data type is a data type that is specific to SQL Server and occupies either 4 or 8 bytes, enabling the storage of approximate values with a defined scale. The scale values permitted are from 1 to 53 and the default scale is 53. Even though a range of values is provided for in the syntax, the current SQL Server implementation of the **float** data type is that, if the scale value is from 1 to 24, the scale is implemented as 24. For any larger value, a scale of 53 is used.

Common Errors

A very common error for new developers is to use approximate numeric data types to store values that need to be stored exactly. This causes rounding and processing errors. A “code smell” for identifying programs that new developers have written is a column of numbers that do not exactly add up to the displayed totals. It is common for small rounding errors to creep into calculations; for example, a total that is incorrect by 1 cent in dollar-based or euro-based currencies.

The inappropriate use of numeric data types can cause processing errors.

Look at the following code and decide how many times the **PRINT** statement would be executed:

Look at the following code and decide how many times the **PRINT** statement would be executed:

How Many Times Is PRINT Executed?

```
DECLARE @Counter float;
SET @Counter = 0;
WHILE (@Counter <> 1.0) BEGIN
    SET @Counter += 0.1;
    PRINT @Counter;
END;
```

In fact, this query would never stop running, and would need to be cancelled.

After cancelling the query, if you looked at the output, you would see the following code:

Resultset from Previous Code Fragment

```
0.1  
0.2  
0.3  
0.4  
0.5  
0.6  
0.7  
0.8  
0.9  
1  
1.1  
1.2  
1.3  
1.4  
1.5  
1.6  
1.7  
...
```

What has happened? The problem is that the value 0.1 cannot be stored exactly in a **float** or **real** data type, so the termination value of the loop is never hit exactly. If a **decimal** value had been used instead, the loop would have executed as expected.

Consider how you would write the answer to $1 \div 3$ in decimal form. The answer isn't 0.3, it is 0.3333333 recurring. There is no way in decimal form to write $1 \div 3$ as an exact decimal fraction. You have to eventually settle for an approximate value.

The same problem occurs in binary fractions; it just occurs at different values—0.1 ends up being stored as the equivalent of 0.099999 recurring. 0.1 in decimal form is a nonterminating fraction in binary. Therefore, when you put the system in a loop adding 0.1 each time, the value never exactly equals 1.0, which can be stored precisely.

Date and Time Data Types

SQL Server supports a rich set of data types for working with values that are related to dates and times. SQL Server also provides a large number of functions for working with dates and times.

date and time Data Types

The **date** data type complies with the ANSI Structured Query Language (SQL) standard definition for the Gregorian calendar. The default string format is YYYY-MM-DD. This format is the same as the ISO 8601 definition for **DATE**. **date** has a range of values from 0001-01-01 to 9999-12-31 with an accuracy of one day.

- Rich set of options is available for storing date and time data
- ISO standard date formats remove ambiguity in date formats, for example 2016-06-01
- Large set of functions available for processing date and time data types

The time data type is aligned to the SQL standard form of hh:mm:ss, with optional decimal places up to hh:mm:ss.nnnnnnnn. Note that when you are defining the data type, you need to specify the number of decimal places, such as **time(4)**, if you do not want to use the default value of seven decimal places, or if

you want to save some storage space. The format that SQL Server uses is similar to the ISO 8601 definition for **TIME**.

The ISO 8601 standard makes it possible to use 24:00:00 to represent midnight and to have a leap second over 59. These are not supported in the SQL Server implementation.

The **datetime2** data type is a combination of a **date** data type and a **time** data type.

datetime Data Type

The **datetime** data type is an older data type that has a smaller range of allowed dates and a lower precision or accuracy. It is a commonly used data type, particularly in older Transact-SQL code. A common error is not allowing for the 3 milliseconds accuracy of the data type. For example, using the **datetime** data type, executing the following code would cause the value '20110101 00:00:00.000' to be stored:

```
DECLARE @When datetime;
SET @When = '20101231 23:59:59.999';
```

Another problem with the **datetime** data type is that the way it converts strings to dates is based on language format settings. A value in the form "YYYYMMDD" will always be converted to the correct date, but a value in the form "YYYY-MM-DD" might end up being interpreted as "YYYY-DD-MM," depending on the settings for the session.

It is important to understand that this behavior does not happen with the new **date** data type, so a string that was in the form "YYYY-MM-DD" could be interpreted as two different dates by the **date** (and **datetime2**) data type and the **datetime** data type. You should specifically check any of the formats that you intend to use, or always use formats that cannot be misinterpreted. Another option that was introduced in SQL Server 2012 can help. A series of functions that enable date and time values to be created from component parts was introduced. For example, there is now a **DATEFROMPARTS** function that you can use to create a date value from a year, a month, and a day.

 **Note:** Be careful when working with string literal representations of dates and time, as they can be interpreted in different ways depending on the location. For example, 01/06/2016 might be June 1, 2016, or the January 6, 2016.

You can solve this ambiguity by expressing dates and times according to the ISO standard, where dates are represented as YYYY-MM-DD. In the previous example, this would be 2016-06-01.

Time Zones

The **datetimeoffset** data type is a combination of a **datetime2** data type and a time zone offset. Note that the data type is not aware of the time zone; it can simply store and retrieve time zone values.

Note that the time zone offset values extend for more than a full day (a range of -14:00 to +14:00). A range of system functions has been provided for working with time zone values, and for all of the data types related to dates and times.

For more information about data and time data type, see Microsoft Docs:

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

<https://aka.ms/Yw0h3v>

For more information about using data and time data types, see Technet:

Using Date and Time Data

<http://go.microsoft.com/fwlink/?LinkId=209249>

Unique Identifiers

Globally unique identifiers (GUIDs) have become common in application development. They are used to provide a mechanism where any process can generate a number and know that it will not clash with a number that any other process has generated.

GUIDs

Numbering systems have traditionally depended on a central source for the next value in a sequence, to make sure that no two processes use the same value. GUIDs were introduced to avoid the need for anyone to function as the "number allocator." Any process (on any system) can generate a value and know that it will not clash with a value generated by any process across time and space, and on any system to an extremely high degree of probability.

This is achieved by using extremely large values. When discussing the **bigint** data type earlier, you learned that the 64-bit **bigint** values were really large. GUIDs are 128-bit values. The magnitude of a 128-bit value is well beyond our capabilities of comprehension.

uniqueidentifier Data Type

The **uniqueidentifier** data type in SQL Server is typically used to store GUIDs. Standard arithmetic operators such as =, <> (or !=), <, >, <=, and >= are supported, in addition to NULL and NOT NULL checks.

The **IDENTITY** property is used to automatically assign values to columns. (**IDENTITY** is discussed in Module 3.) The **IDENTITY** property is not used with **uniqueidentifier** columns. New values are not calculated by code in your process. They are calculated by calling system functions that generate a value for you. In SQL Server, this function is the **NEWID()** function.

The random nature of GUIDs has also caused significant problems in current storage subsystems. SQL Server 2005 introduced the **NEWSEQUENTIALID()** function to try to circumvent the randomness of the values that the **NEWID()** function generated. However, the function does so at the expense of some guarantee of uniqueness.

The usefulness of the **NEWSEQUENTIALID()** function is limited because the main reason for using GUIDs is to enable other layers of code to generate the values, and know that they can just insert them into a database without clashes. If you need to request a value from the database via **NEWSEQUENTIALID()**, it usually would have been better to use an **IDENTITY** column instead.

A common development error is to store GUIDs in string values rather than in **uniqueidentifier** columns.

- **uniqueidentifier** data type is typically used for storing GUID values
- GUID stands for globally unique identifier
- Storage is essentially a 128-bit integer, but standard integer arithmetic is not supported
- =, <>, <, >, <=, >= are supported along with NULL and NOT NULL checking
- **IDENTITY** cannot be used
- New values from **NEWID()** function
- Common error is to store GUIDs as strings



Note: Replication systems also commonly use **uniqueidentifier** columns. Replication is an advanced topic that is beyond the scope of this course.

NULL and NOT NULL

Nullability determines whether or not a value must be entered. For example, a column constraint might allow NULLs, or not allow NULLs by specifying NOT NULL. Allowing NULLs when values should be entered, is another common design error.

NULL

NULL is a state of a column in a particular row, rather than a type of value that is stored in a column. You do not say that a value equals NULL; you say that a value is NULL. This is why, in Transact-SQL, you do not check whether a value is NULL with the equality operator. For example, you would not write the following code:

```
WHERE Color = NULL;
```

Instead, you would write the following code:

```
WHERE Color IS NULL;
```

Common Errors

New developers often confuse NULL values with zero, blank (or space), zero-length strings, and so on. The misunderstanding is exacerbated by other database engines that treat NULL and zero-length strings or zeroes as identical. NULL indicates the absence of a value.

Careful consideration must be given to the nullability of a column. In addition to specifying a data type for a column, you specify whether a value needs to be present. Often, this is referred to as whether a column value is mandatory.

Look at the NULL and NOT NULL declarations in the following code sample and decide why each decision might have been made:

Look at the NULL and NOT NULL declarations in the following code sample and decide why each decision might have been made:

NULL or NOT NULL?

```
CREATE TABLE Sales.Opportunity
(
    OpportunityID int NOT NULL,
    Requirements nvarchar(50) NOT NULL,
    ReceivedDate date NOT NULL,
    LikelyClosingDate date NULL,
    SalesPersonID int NULL,
    Rating int NOT NULL
);
```

- NULL or NOT NULL determines whether or not a value must be provided
- NULL indicates the absence of a value
- Can be defined on columns and parameters
- Cannot be defined on variables
- SET ANSI_NULL_DFLT_ON—may be set on or off

For more information about allowing NULL values, see Technet:

Allowing Null Values

<http://go.microsoft.com/fwlink/?LinkId=209251>

You can set the default behavior for new columns using the ANSI NULL default. For details about how this works, see MSDN:

SET ANSI_NULL_DFLT_ON (Transact-SQL)

<http://go.microsoft.com/fwlink/?LinkId=233793>

Alias Data Types

An alias data type is based on a system type, and is useful when numerous tables within a database share a similar column type. For example, a retail database contains a table for **Store**, **Supplier**, **Customer**, and **Employee**: each of these tables contains address columns, including postal code. You could create an alias data type to set the required format of the post code column, and use this type for all **PostCode** columns in each of the tables. Create a new alias data type using the **CREATE TYPE** command. This creates the user type in the current database only:

```
CREATE TYPE dbo.PostalCode
FROM varchar(8);
```

- An alias data type is created using the **CREATE TYPE** command
- The alias data type is created in the context of the current database
- Create alias data types in the model database to automatically create a user type in all future databases
- Query **sys.types** to view alias data types for a database
- Has same use as system types: set column, variable, and parameter data types with alias

After creating an alias data type, it is used in the same way as a system data type. The alias data type is used as the data type when creating a column. In the following code, the **PostCode** column uses the new **PostalCode** data type. There is no need to specify the width of the column because this was done when the type was created.

Example of using an alias data type:

Example of using an alias data type:

PostalCode Is an Alias Data Type

```
CREATE TABLE dbo.Store
(
    StoreID int IDENTITY(1,1) PRIMARY KEY,
    StoreName nvarchar(30) NOT NULL,
    Address1 nvarchar(30) NOT NULL,
    Address2 nvarchar(30) NULL,
    City nvarchar(25) NOT NULL,
    PostCode PostalCode NOT NULL
);
```

When declaring variables or parameters, the data type assignment is again used in exactly the same way as system data types:

```
DECLARE @PostCode AS PostalCode
```

To discover which alias data types have already been created within a database, query the **sys.types** system view within the context of the relevant database.



Note: You can create alias data types in the model database, so every time a new database is created, the user data types will automatically be created.

Converting Data Between Data Types

When converting between data types, there is some choice in deciding which conversion function to use, and some rules for others. The CAST and CONVERT functions can be used for all data types, whereas TRY_PARSE should only be used when converting from string to date and/or time, or for numeric values.

CAST and CONVERT

The CAST and CONVERT functions are similar; however, you should use CAST if your code needs to conform to SQL-92 standardization. CONVERT is more useful for datetime conversions, as it offers a range of formats for converting datetime expressions.

The CAST function accepts an expression for the first parameter then, after the AS keyword, the data type to which the expression should be converted. The following example converts today's date to a string. There is no option to format the date, so using CONVERT would be a better choice:

Example of using CAST:

Example of using CAST:

CAST

```
SELECT CAST(GETDATE() AS nvarchar(50)) AS DateToday;
DateToday
-----
Jan 25 2016  3:21PM
```

CAST will try to convert one data type to another.

The following code passes a string with a whole number, so SQL Server can easily convert this to an integer type:

The following code passes a string with a whole number, so SQL Server can easily convert this to an integer type:

CAST

```
SELECT CAST('93751' AS int) AS ConvertedString;
ConvertedString
-----
93751
```

- CAST and CONVERT are similar
 - CAST conforms to SQL-92 standards
 - Use CONVERT for formatting datetime expressions
- PARSE
 - Similar to CAST, but accepts a culture parameter
- TRY_CAST
 - Same as CAST, but returns NULL instead of an error
- TRY_CONVERT
 - Same as CONVERT, but returns NULL instead of an error
- TRY_PARSE
 - Same as PARSE, but returns NULL instead of an error

If the string is changed to a decimal, an error will be thrown:

CAST

```
SELECT CAST('93751.3' AS int) AS ConvertedString;  
  
Msg 245, Level 16, State 1, Line 4  
Conversion failed when converting the varchar value '93751.2' to data type int.
```

In the following code, CONVERT accepts three parameters—the data type to which the expression should be converted, the expression, and the datetime format for conversion:

In the following code, CONVERT accepts three parameters—the data type to which the expression should be converted, the expression, and the datetime format for conversion:

CONVERT

```
SELECT CONVERT(nvarchar(50), GETDATE(), 106) AS DateToday;  
  
DateToday  
-----  
25 Jan 2016
```

PARSE

The structure of the PARSE function is similar to CAST; however, it accepts an optional parameter through the **USING** keyword that enables you to set the culture of the expression. If no culture parameter is provided, the function will use the language of the current session. PARSE should only be used for converting strings to date/time or numbers, including money.

In the following example, the session language uses the **British English** culture, which uses the date format DD/MM/YYYY. The **US English** date expression is in the American format MM/DD/YYYY, and is parsed into the British English language:

In the following example, the session language uses the **British English** culture, which uses the date format DD/MM/YYYY. The **US English** date expression is in the American format MM/DD/YYYY, and is parsed into the British English language:

PARSE

```
SET LANGUAGE 'British English';  
SELECT PARSE('10/13/2015' AS datetime2 USING 'en-US') AS MyDate;  
  
MyDate  
-----  
2015-10-13 00:00:00.0000000
```

If the optional parameter is excluded, then the parser will try to convert the date, and throw an error:

If the optional parameter is excluded, then the parser will try to convert the date, and throw an error:

PARSE

```
SET LANGUAGE 'British English';  
SELECT PARSE('10/13/2015' AS datetime2) AS MyDate;  
  
-----  
Msg 9819, Level 16, State 1, Line 7  
Error converting string value '10/13/2015' into data type datetime2 using culture ''.
```

To find out which languages are present on an instance of SQL Server, run the following code:

```
SELECT * FROM sys.syslanguages
```

Use the **alias** column value in the SET LANGUAGE statement:

```
SET LANGUAGE 'Turkish';
```

TRY_CAST and TRY_CONVERT

TRY_CAST operates in much the same way as CAST, but will return NULL rather than an error, if the expression cannot be cast into the intended data type.

The following query executes the code used in the above CAST example that failed, but this time returns NULL:

TRY_CAST

```
SELECT TRY_CAST('93751.3' AS int) AS ConvertedString;
ConvertedString
-----
NULL
```

This is useful for eloquently handling errors in your code, as a NULL is easier to deal with than an error message.

It can also be used with the CASE statement, as per the following example:

TRY_CAST

```
SELECT
CASE
    WHEN TRY_CAST('93751.3' AS int) IS NULL THEN 'FAIL'
    ELSE 'SUCCESS'
END AS ConvertedString;
ConvertedString
-----
FAIL
```

Just as TRY_CAST is similar to CAST, TRY_CONVERT works the same as CONVERT but returns NULL instead of an error when an expression cannot be converted. It, too, can also be used in a CASE statement.

Example of using TRY_CONVERT:

TRY_CONVERT

```
SELECT
CASE
    WHEN TRY_CONVERT(varchar(25), 93751.3) IS NULL THEN 'FAIL'
    ELSE 'SUCCESS'
END AS ConvertedString;
ConvertedString
-----
SUCCESS
```

TRY_PARSE

Following the format of the previous TRY functions, TRY_PARSE is identical to PARSE, but returns a NULL instead of an error when an expression cannot be parsed.

When using TRY_PARSE, and running the code sample that failed, A NULL is returned, rather than an error:

TRY_PARSE

```
SET LANGUAGE 'British English';
SELECT TRY_PARSE('10/13/2015' AS datetime2) AS MyDate;

MyDate
-----
NULL
```

If you use CAST, CONVERT or PARSE in your application code and the parser throws an error that you haven't handled, it may cause issues in the application. Use TRY_CAST, TRY_CONVERT and TRY_PARSE when you need to handle an error, and use the CASE statement to provide alternative values when a conversion is not possible.

SQL Server Data Type Conversion Chart

<http://aka.ms/xd82ey>

Working with International Character Data

Traditionally, most computer systems stored one character per byte. This only allowed for 256 different character values, which is not enough to store characters from many languages.

Multibyte Character Issues

Asian languages, such as Chinese and Japanese, need to store thousands of characters. You may not have ever considered it, but how would you type these characters on a keyboard? There are two basic ways that this is accomplished. One option is to have an English-like version of the language that can be used for entry. Japanese has a language form called Romaji that uses English-like characters for representing words. Chinese has a form called Pinyin that is also somewhat English-like.

Users can enter the number beside the character to select the intended word. It might not seem important to an English-speaking person but, given that the first option means "horse", the second option is like a question mark, and the third option means "mother", there is definitely a need to select the correct option!

- Unicode
 - Is a worldwide character-encoding standard
 - Simplifies software localization
 - Improves multilingual character processing
 - Is implemented in SQL Server as double-byte for Unicode types
 - Requires N prefix
 - Uses LEN() to return number of characters
 - Uses DATALENGTH() to return the number of bytes

Character Groups

An alternate way to enter the characters is via radical groupings.

Character Group Example

```
DECLARE @Hello nvarchar(10);

SET @Hello = N'Hello';
SET @Hello = N'你好';
SET @Hello = N'こんにちは';
```

Note the third character in the preceding code example. The left-hand part of that character, 女, means "woman". Rather than entering English-like characters (that could be quite unfamiliar to the writers), select a group of characters based on what is known as a radical.

Note that the character representing "mother" is the first character on the second line. For this sort of keyboard entry to work, the characters must be in appropriate groups, not just stored as one large sea of characters. An additional complexity is that the radicals themselves are also in groups. In the screenshot, you can see that the woman radical was part of the third group of radicals.

Unicode

In the 1980s, work was done by a variety of researchers, to determine how many bytes are required to be able to hold all characters from all languages, but also store them in their correct groupings. The answer from all researchers was three bytes. You can imagine that three was not an ideal number for computing and at the time, users were mostly working with 2 byte (that is, 16-bit) computer systems.

Unicode introduced a two-byte character set that attempts to fit the values from the 3 bytes into 2 bytes. Inevitably, there had to be some trade-offs.

Unicode allows any combination of characters, which are drawn from any combination of languages, to exist in a single document. There are multiple encodings for Unicode with UTF-7, UTF-8, UTF-16, and UTF-32. (UTF is universal text format.) SQL Server currently implements double-byte UTF-16 characters for its Unicode implementation.

For string literal values, an N prefix on a string allows the entry of double-byte characters into the string, rather than just single-byte characters. (N stands for "National" in National Character Set.)

When working with character strings, the LEN function returns the number of characters (Unicode or not) whereas DATALENGTH returns the number of bytes.

Question: What would be a suitable data type for storing the value of a check box that can be 0 for cleared, 1 for selected, or -1 for disabled?

Lesson 3

Working with Schemas

A schema is a namespace that allows objects within a database to be logically separated to make them easier to manage. Objects may be separated according to the owner, according to their function, or any other way that makes sense for a particular database.

Schemas were introduced with SQL Server 2005. They can be thought of as containers for objects such as tables, views, and stored procedures. Schemas provide organization and structure when a database includes large numbers of objects.

You can also assign security permissions at the schema level, rather than for individual objects that are contained within the schemas. Doing this can greatly simplify the design of system security requirements.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the role of a schema.
- Understand object name resolution.
- Create schemas.

What Is a Schema?

Schemas are used to contain objects and to provide a security boundary for the assignment of permissions. In SQL Server, schemas are used as containers for objects, rather like a folder is used to hold files at the operating system level. Since their introduction in SQL Server 2005, schemas can be used to contain objects such as tables, stored procedures, functions, types, and views. Schemas form a part of the multipart naming convention for objects. In SQL Server, an object is formally referred to by a name of the form

Server.Database.Schema.Object.

Security Boundary

Schemas can be used to simplify the assignment of permissions. An example of applying permissions at the schema level would be to assign the EXECUTE permission on a schema to a user. The user could then execute all stored procedures within the schema. This simplifies the granting of permissions because there is no need to set up individual permissions on each stored procedure.

It is important to understand that schemas are not used to define physical storage locations for data, as occurs in some other database engines.

Upgrading Older Applications

If you are upgrading applications from SQL Server 2000 and earlier versions, it is important to understand that the naming convention changed when schemas were introduced. Previously, names were of the form ***Server.Database.Owner.Object.***

- Schemas are containers for objects such as:
 - Tables
 - Stored procedures
 - Functions
 - Types
 - Views
- Schemas are security boundaries
 - Permissions can be granted at the schema level to apply to all objects within a schema
 - Simplifies security configuration

Objects still have owners, but the owner's name does not form a part of the multipart naming convention from SQL Server 2005 onward. When upgrading databases from earlier versions, SQL Server will automatically create a schema that has the same name as existing object owners, so that applications that use multipart names will continue to work.

Object Name Resolution

It is important to use at least two-part names when referring to objects in SQL Server code, such as stored procedures, functions, and views.

Object Name Resolution

When object names are referred to in the code, SQL Server must determine which underlying objects are being referred to. For example, consider the following statement:

```
SELECT ProductID, Name, Size FROM  
Product;
```

- If the schema name is omitted, rules apply for name resolution
 - Users can have a default schema assigned
 - Users who have no default schema will have dbo as their default schema
 - Initially, the user's default schema is searched
 - If the object is not found in the default schema, the dbo schema is also searched
- When referencing an object in a statement, users should specify both the schema and the object name
 - Select ProductID FROM Production.Product;

More than one Product table could exist in separate schemas of the same database. When single-part names are used, SQL Server must then determine which Product table is being referred to.

Most users have default schemas assigned, but not all types of users have these. Default schemas are assigned to users based on standard Windows® and SQL Server logins. You can also assign default schemas to Windows groups when using SQL Server 2012. Users without default schemas are considered to have the dbo schema as their default schema.

When locating an object, SQL Server will first check the user's default schema. If the object is not found, SQL Server will then check the dbo schema to try to locate it.

It is important to include schema names when referring to objects, instead of depending upon schema name resolution, such as in this modified version of the previous statement:

```
SELECT ProductID, Name, Size FROM Production.Product;
```

Apart from rare situations, using multipart names leads to more reliable code that does not depend upon default schema settings.

Creating Schemas

Schemas are created by using the **CREATE SCHEMA** command. This command can also include the definition of objects to be created within the schema at the time the schema is created.

CREATE SCHEMA

Schemas have both names and owners. In the first example shown on the slide, a schema named Reporting is being created. It is owned by the user, Terry. Although both schemas and the objects contained in the schemas have owners and the owners do not have to be the same, having different owners for schemas and the objects contained within them can lead to complex security issues.

- Schemas are created by using the CREATE SCHEMA command
- Schemas have owners
 - Objects contained within schemas also have owners
 - Specify objects and permissions when the schema is created

Object Creation at Schema Creation Time

Besides creating schemas, the **CREATE SCHEMA** statement can include options for object creation. Although the code example that follows might appear to be three statements (**CREATE SCHEMA**, **CREATE TABLE**, and **GRANT**), it is in fact a single statement. Both **CREATE TABLE** and **GRANT** are options that are being applied to the **CREATE SCHEMA** statement.

Within the newly created KnowledgeBase schema, the Article table is being created and the SELECT permission on the database is being granted to **Salespeople**.

Statements such as the second **CREATE SCHEMA** statement can lead to issues if the entire statement is not executed together.

Object creation when the schema is created.

CREATE SCHEMA

```

CREATE SCHEMA Reporting
AUTHORIZATION Terry;

CREATE SCHEMA KnowledgeBase
AUTHORIZATION Paul;

CREATE TABLE Article
(
    ArticleID int IDENTITY (1,1) PRIMARY KEY,
    Articlecontents XML
);

GRANT SELECT TO SalesPeople;

```

Demonstration: Working with Schemas

In this demonstration, you will see how to:

- Create a schema.
- Create a schema with an included object.
- Drop a schema.

Demonstration Steps

1. Ensure that you have completed the previous demonstrations in this module.
2. On the taskbar, click **Microsoft SQL Server Management Studio**.
3. In the **Connect to Server** dialog box, in the **Server** box, type the URL of the Azure server <Server Name>.database.windows.net (where <Server Name> is the name of the server you created).
4. In the **Authentication** list, click **SQL Server Authentication**.
5. In the **User name** box, type **Student**, and in the **Password** box, type **Pa55w.rd**, and then click **Connect**.
6. On the **File** menu, point to **Open**, click **Project/Solution**.
7. In the **Open Project** dialog box, navigate to the **D:\Demofiles\Mod02** folder, click **Demo.ssmssln**, and then click **Open**.
8. In Solution Explorer, under **Queries**, double-click **2 - Schemas.sql**.
9. In the **Available Databases** list, click **AdventureWorksLT**.
10. Select the code under the **Step 2: Create a Schema** comment, and then click **Execute**.
11. Select the code under the **Step 3: Create a table using the new schema** comment, and then click **Execute**.
12. Select the code under the **Step 4: Drop the schema** comment, and then click **Execute**. Note that the schema cannot be dropped while objects exist in it.
13. Select the code under the **Step 5: Drop and the table and then the schema** comment, and then click **Execute**.
14. Leave SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Question
Which of the following objects cannot be stored in a schema?
Select the correct answer.
<input type="checkbox"/> Table
<input type="checkbox"/> Function
<input type="checkbox"/> Database role
<input type="checkbox"/> View
<input type="checkbox"/> Stored procedure

Lesson 4

Creating and Altering Tables

Now that you understand the core concepts surrounding the design of tables, this lesson introduces you to the Transact-SQL syntax that is used when defining, modifying, or dropping tables. Temporary tables are a special form of table that can be used to hold temporary result sets. Computed columns are used to create columns where the value held in the column is automatically calculated, either from expressions involving other columns from the table, or from the execution of functions.

Lesson Objectives

After completing this lesson, you will be able to:

- Create tables
- Drop tables
- Alter tables
- Use temporary tables
- Work with computed columns

Creating Tables

Tables are created by using the **CREATE TABLE** statement. This statement is also used to define the columns that are associated with the table, and identify constraints such as primary and secondary keys.

CREATE TABLE

When you create tables by using the **CREATE TABLE** statement, make sure that you supply both a schema name and a table name. If the schema name is not specified, the table will be created in the default schema of the user who is executing the statement. This could lead to the creation of scripts that are not robust, because they could generate different schema designs when different users execute them.

- Tables are created using the CREATE TABLE statement
- Specify column names and data types
- Specify NULL or NOT NULL
- Specify the primary key

Nullability

You should specify NULL or NOT NULL for each column in the table. SQL Server has defaults for this that you can change via the **ANSI_NULL_DEFAULT** setting. Scripts should always be designed to be as reliable as possible—specifying nullability in data definition language (DDL) scripts helps to improve script reliability.

Primary Key

You can specify a primary key constraint beside the name of a column if only a single column is included in the key. It must be included after the list of columns when more than one column is included in the key.

In the following example, the **SalesID** value is only unique for each **SalesRegisterID** value:

Specifying a Primary Key

```
CREATE TABLE PetStore.SalesReceipt
(
    SalesRegisterID int NOT NULL,
    SalesID int NOT NULL,
    CustomerID int NOT NULL,
    SalesAmount decimal(18,2) NOT NULL,
    PRIMARY KEY (SalesRegisterID, SalesID)
);
```

Primary keys are constraints and are more fully described, along with other constraints, later in this course.

Dropping Tables

The **DROP TABLE** statement is used to delete a table from a database. If a table is referenced by a foreign key constraint, it cannot be dropped.

When dropping a table, all permissions, constraints, indexes, and triggers that are related to the table are also dropped. Deletion is permanent. SQL Server has no equivalent to the Windows Recycle Bin—after the table is dropped, it is permanently removed.

Code that references the table, such as code that is contained within stored procedures, functions, and views, is not dropped. This can lead to “orphaned” code that refers to nonexistent objects. SQL Server 2008 introduced a set of dependency views that can be used to locate code that references nonexistent objects. The details of both referenced and referencing entities are available from the **sys.sql_expression_dependencies** view. Referenced and referencing entities are also available separately from the **sys.dm_sql_referenced_entities** and **sys.dm_sql_referencing_entities** dynamic management views. Views are discussed later in this course.

Using the DRO statement.

DROP

```
DROP TABLE PetStore.Owner;
GO
```

- Tables are removed by using the **DROP TABLE** statement
- Reference tables (via foreign keys) cannot be dropped
- All permissions, constraints, indexes, and triggers are also dropped
- Code that references the table, such as a stored procedure, is not dropped

Altering Tables

Altering a table is useful because permissions on the table are retained, along with the data in the table. If you drop and recreate the table with a new definition, both the permissions on the table and the data in the table are lost. However, if the table is referenced by a foreign key, it cannot be dropped, though it can be altered.

Tables are modified by using the **ALTER TABLE** statement. You can use this statement to add or drop columns and constraints, or to enable or disable constraints and triggers. Constraints and triggers are discussed in later modules.

- Use the ALTER TABLE statement to modify tables
- ALTER TABLE retains permissions to the table
- ALTER TABLE retains the data in the table
- ALTER TABLE is used to:
 - Add or drop columns and constraints
 - Enable or disable constraints and triggers

Note that the syntax for adding and dropping columns is inconsistent. The word **COLUMN** is required for **DROP**, but not for **ADD**. In fact, it is not an optional keyword for **ADD** either. If the word **COLUMN** is omitted in a **DROP**, SQL Server assumes that it is a constraint being dropped.

In the following example, the **PreferredName** column is being added to the PetStore.Owner table. Then, the **PreferredName** column is being dropped from the PetStore.Owner table. Note the difference in syntax regarding the word **COLUMN**.

Use ALTER TABLE to add or delete columns.

ALTER TABLE

```
ALTER TABLE Petstore.Owner
ADD PreferredName nvarchar(30) NULL;
GO

ALTER TABLE Petstore.Owner
DROP COLUMN PreferredName;
GO
```

Demonstration: Working with Tables

In this demonstration, you will see how to:

- Create tables and alter tables.
- Drop tables.

Demonstration Steps

1. Ensure that you have completed the previous demonstrations in this module.
2. In SQL Server Management Studio, in Solution Explorer, under **Queries**, double-click **3 - Create Tables.sql**.
3. In the **Available Databases** list, click **AdventureWorksLT**.
4. Select the code under the **Step 2: Create a table** comment, and then click **Execute**.
5. Select the code under the **Step 3: Alter the SalesLT.Courier table** comment, and then click **Execute**.
6. Select the code under the **Step 4: Drop the tables** comment, and then click **Execute**.
7. Leave SQL Server Management Studio open for the next demonstration

Temporary Tables

Temporary tables are used to hold temporary result sets within a user's session. They are created within the **tempdb** database and deleted automatically when they go out of scope. This typically occurs when the code in which they were created completes or aborts. Temporary tables are very similar to other tables, except that they are only visible to the creator and in the same scope (and subscopes) within the session. They are automatically deleted when a session ends or when they go out of scope. Although temporary tables are deleted when they go out of scope, you should explicitly delete them when they are no longer needed to reduce resource requirements on the server. Temporary tables are often created in code by using the **SELECT INTO** statement.

- Session temporary tables are only visible to their creators in the same session and same scope or subscope
 - Created with # prefix
 - Dropped when the user disconnects or when out of scope
 - Should be deleted in code rather than depending on automatic drop
 - Often created by using SELECT INTO statements
- Global temporary tables are visible to all users
 - Created with ## prefix
 - Deleted when all users referencing the table disconnect

A table is created as a temporary table if its name has a number sign (#) prefix. A global temporary table is created if the name has a double number sign (##) prefix. Global temporary tables are visible to all users and are not commonly used.

Passing Temporary Tables

Temporary tables are also often used to pass rowsets between stored procedures. For example, a temporary table that is created in a stored procedure is visible to other stored procedures that are executed from within the first procedure. Although this use is possible, it is not considered good practice in general. It breaks common rules of abstraction for coding and also makes it more difficult to debug or troubleshoot the nested procedures. SQL Server 2008 introduced table-valued parameters (TVPs) that can provide an alternate mechanism for passing tables to stored procedures or functions. (TVPs are discussed later in this course.)

The overuse of temporary tables is a common Transact-SQL coding error that often leads to performance and resource issues. Extensive use of temporary tables can be an indicator of poor coding techniques, often due to a lack of set-based logic design.

Demonstration: Working with Temporary Tables

In this demonstration, you will see how to:

- Create local temporary tables.
- Create global temporary tables.
- Access a global temporary table from another session.

Demonstration Steps

1. Ensure that you have completed the previous demonstrations in this module.
2. In SQL Server Management Studio, in Solution Explorer, under **Queries**, double-click **4 - Temporary Tables.sql**.
3. Right-click the query pane, point to **Connection**, and then click **Change Connection**.
4. In the **Connect to Database Engine window** dialog box, in the **Server name** box, type **MIA-SQL**, in the **Authentication** box, select **Windows Authentication**, and then click **Connect**.
5. Select the code under the **Step 1: Create a local temporary table** comment, and then click **Execute**.
6. In Solution Explorer, under **Queries**, double-click **5 - Temporary Tables.sql**.
7. Select the code under the **Step 1: Select and execute the following query** comment, and then click **Execute**. Note that this session cannot access the local temporary table from the other session.
8. Switch to the **4 - Temporary Tables.sql** pane.
9. Select the code under the **Step 3: Create a global temporary table** comment, and then click **Execute**.
10. Switch to the **5 - Temporary Tables.sql** pane.
11. Select the code under the **Step 2: Select and execute the following query** comment, and then click **Execute**. Note that this session can access the global temporary table from the other session.
12. Switch to the **4 - Temporary Tables.sql** pane.
13. Select the code under the **Step 5: Drop the two temporary tables** comment, and then click **Execute**.
14. Leave SQL Server Management Studio open for the next demonstration

Computed Columns

Computed columns are derived from other columns or from the result of executing functions.

Computed columns were introduced in SQL Server 2000. In this example, the **YearOfBirth** column is calculated by executing the **DATEPART** function to extract the year from the **DateOfBirth** column in the same table.

You can also see the word **PERSISTED** added to the definition of the computed column. Persisted computed columns were introduced in SQL Server 2005.

Defining a persisted computer column.

- Computed columns are derived from other columns or functions
- Computed columns are often used to provide easier access to data without denormalizing it
- Persisted computed columns improve SELECT performance of computed columns in some situations

Computed Column

```
CREATE TABLE PetStore.Pet
(
    PetID int IDENTITY (1,1) PRIMARY KEY,
    PetName nvarchar(30) NOT NULL,
    DateOfBirth date NOT NULL,
    YearOfBirth AS DATEPART(year, DateOfBirth) PERSISTED
);
GO
```

A nonpersisted computed column is calculated every time a **SELECT** operation occurs on the column and it does not consume space on disk. A persisted computed column is calculated when the data in the row is inserted or updated and does consume space on the disk. The data in the column is then selected like the data in any other column.

The difference between persisted and nonpersisted computed columns relates to when the computational performance impact is exerted.

- Nonpersisted computed columns work best for data that is modified regularly, but rarely selected.
- Persisted computed columns work best for data that is modified rarely, but selected regularly.
- In most business systems, data is read much more regularly than it is updated. For this reason, most computed columns would perform best as persisted computed columns.

Demonstration: Working with Computed Columns

In this demonstration, you will see how to:

- Work with computed columns.
- Use PERSISTED columns.

Demonstration Steps

1. Ensure that you have completed the previous demonstrations in this module.
2. In SQL Server Management Studio, in Solution Explorer, under **Queries**, double-click **6 - Computed Columns.sql**.
3. Right-click the query pane, point to **Connection**, and then click **Change Connection**.
4. In the **Connect to Database Engine window** dialog box, in the **Server name** box, type the URL for the Azure account, in the **Authentication** box, select **SQL Server Authentication**, in the **Login** box, type **Student**, and in the **Password** box, type **Pa55w.rd**, and then click **Connect**.
5. In the **Available Databases** list, click **AdventureWorksLT**.
6. Select the code under the **Step 2: Create a table with two computed columns** comment, and then click **Execute**.
7. Select the code under the **Step 3: Populate the table with data** comment, and then click **Execute**.
8. Select the code under the **Step 4: Return the results from the SalesLT.SalesOrderDates table** comment, and then click **Execute**.
9. Select the code under the **Step 5: Update a row in the SalesLT.SalesOrderDates table** comment, and then click **Execute**.
10. Select the code under the **Step 6: Create a table with a computed column that is not persisted** comment, and then click **Execute**.
11. Select the code under the **Step 7: Populate the table with data** comment, and then click **Execute**.
12. Select the code under the **Step 8 - Return the results from the SalesLT.TotalSales table** comment, and then click **Execute**.
13. Close SQL Server Management Studio without saving any changes.

Question: When creating a computed column, why is it good practice to include the PERSISTED keyword? What are the consequences of excluding PERSISTED when the table has several million records?

Lab: Designing and Implementing Tables

Scenario

A business analyst from your organization has given you a draft design for some new tables being added to a database. You need to provide an improved schema design, based on good design practices. After you have designed the schema and tables, you need to implement them in the TSQL database.

Objectives

After completing this lab, you will be able to:

- Choose an appropriate level of normalization for table data.
- Create a schema.
- Create tables.

Estimated Time: 45 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Designing Tables

Scenario

A business analyst from your organization has given you a first pass at a schema design for some new tables being added to the TSQL database. You need to provide an improved schema design, based on good design practices and an appropriate level of normalization. The business analyst was also confused about when data should be nullable. You need to decide about nullability for each column in your improved design.

The main tasks for this exercise are as follows:

1. Prepare the Environment
2. Review the Design
3. Improve the Design

► Task 1: Prepare the Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab02\Starter** folder as Administrator.

► Task 2: Review the Design

1. Open the **Schema Design for Marketing Development Tables.docx** from the **D:\Labfiles\Lab02\Starter** folder.
2. Review the proposed structure for the new tables.

► Task 3: Improve the Design

1. Complete the **Allow Nulls?** column for each table.
2. Save your document.

3. Review the suggested solution in **Schema Design for Marketing Development Tables.docx** in the **D:\Labfiles\Lab02\Solution** folder.
4. Close WordPad.

Results: After completing this exercise, you will have an improved schema and table design.

Exercise 2: Creating Schemas

Scenario

The new tables will be isolated in their own schema. You need to create the required schema called DirectMarketing and assign ownership to the dbo user.

The main tasks for this exercise are as follows:

1. Create a Schema

► Task 1: Create a Schema

1. Using SSMS, connect to MIA-SQL using Windows Authentication.
2. Open **Project.ssmssln** from the **D:\Labfiles\Lab02\Starter\Project** folder.
3. In the **Lab Exercise 2.sql** file, write and execute a query to create the **DirectMarketing** schema, and set the authorization to the **dbo** user.

Results: After completing this exercise, you will have a new schema in the database.

Exercise 3: Creating Tables

Scenario

You need to create the tables that you designed earlier in this lab. You should use appropriate nullability for each column and each table should have a primary key. At this point, there is no need to create CHECK or FOREIGN KEY constraints.

The main tasks for this exercise are as follows:

1. Create the Competitor Table
2. Create the TVAdvertisement Table
3. Create the CampaignResponse Table

► Task 1: Create the Competitor Table

1. In the Lab Exercise 3.sql file, write and execute a query to create the **Competitor** table that you designed in Exercise 1 in the **DirectMarketing** schema.
2. In Object Explorer, verify that the new table exists.

► Task 2: Create the TVAdvertisement Table

1. In the Lab Exercise 3.sql file, write and execute a query to create the **TVAdvertisement** table that you designed in Exercise 1 in the **DirectMarketing** schema.
2. Refresh Object Explorer and verify that the new table exists.

► **Task 3: Create the CampaignResponse Table**

1. In the Lab Exercise 3.sql file, write and execute a query to create the **CampaignResponse** table that you designed in Exercise 1 in the **DirectMarketing** schema.
2. Refresh Object Explorer and verify that the new table exists.
3. Review the **Computed text** property of the **ResponseProfit** column.
4. Close SQL Server Management Studio without saving changes.

Results: After completing this exercise you will have created the Competitor, TVAdvertisement, and the CampaignResponse tables. You will have created table columns with the appropriate NULL or NOT NULL settings, and primary keys.

Question: When should a column be declared as nullable?

Module Review and Takeaways



Best Practice: All tables should have primary keys. Foreign keys should be declared within the database in almost all circumstances. Developers often suggest that the application will ensure referential integrity, but experience shows that this is a poor option. Databases are often accessed by multiple applications, and bugs are also easy to miss when they first start to occur.

Module 3

Advanced Table Designs

Contents:

Module Overview	3-1
Lesson 1: Partitioning Data	3-2
Lesson 2: Compressing Data	3-13
Lesson 3: Temporal Tables	3-19
Lab: Using Advanced Table Designs	3-27
Module Review and Takeaways	3-31

Module Overview

The physical design of a database can have a significant impact on the ability of the database to meet the storage and performance requirements set out by the stakeholders. Designing a physical database implementation includes planning the file groups, how to use partitioning to manage large tables, and using compression to improve storage and performance. Temporal tables are a new feature in SQL Server® 2016 and offer a straightforward solution to collecting changes to your data.

Objectives

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

- Describe the considerations for using partitioned tables in a SQL Server database.
- Plan for using data compression in a SQL Server database.
- Use temporal tables to store and query changes to your data.

Lesson 1

Partitioning Data

Databases that contain very large tables are often difficult to manage and might not scale well. This lesson explains how you can use partitioning to overcome these problems, ensuring that databases remain efficient, and can grow in a managed, orderly manner.

Lesson Objectives

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

- Understand common situations where partitioning can be applied.
- Use partition functions.
- Use partition schemas.
- Create a partitioned schema.
- Add indexes to your partitions.
- Understand the SWITCH, MERGE, and SPLIT operations.
- Design a partition strategy.

Common Scenarios for Partitioning

You can use SQL Server to partition tables and indexes. A partitioned object is a table or index that is divided into smaller units based on the values in a particular column, called the partitioning key. Partitioning divides data horizontally—often called sharding—so partitions are formed of groups of rows. Before SQL Server 2016 Service Pack 1, partitioning is only available in the SQL Server Enterprise and Developer Editions. In SQL Server 2016 Service Pack 1 and later editions, partitioning is available in all editions of SQL Server, and also in Azure™ SQL Database.

Partitioning improves manageability for large tables and indexes, particularly when you need to load large volumes of data into tables, or remove data from tables. You can manipulate data in partitioned tables by using a set of dedicated commands that enable you to merge, split and switch partitions. These operations often only move metadata, rather than moving data—which makes tasks such as loading data much faster. They are also less resource intensive than loading data by using INSERTs. In some cases, partitioning can also improve query performance, by moving older data out, thereby reducing the volume of data to be queried.

- Partitioned tables and indexes make managing large tables more efficient
- Tables are partitioned horizontally in rows
- Common scenarios for table partitioning include:
 - **Implementing a sliding window:** move data into and out of tables based on date ranges
 - **Separating maintenance operations:** perform maintenance operations, such as rebuilding and reorganizing indexes on a partition-by-partition basis, which is more efficient than doing it for the whole table
 - **Performing partial backups:** use multiple filegroups to store partitioned tables and indexes
- Available in all editions after SQL 2016 SP1

Common scenarios for partitioning a table include:

- **Implementing a sliding window.** In a sliding window scenario, you move data into and out of tables based on date ranges. For example, in a data warehouse, you could partition large fact tables; you could move data out of a table when it is older than one year and no longer of use to data analysts, and then load newer data to replace it. You will learn more about sliding window scenarios in this lesson.
- **Enabling separate maintenance operations.** You can perform maintenance operations, such as rebuilding and reorganizing indexes on a partition-by-partition basis, which might be more efficient than doing it for the whole table. This is particularly useful when only some of the partitions contain data that changes—because there is no need to maintain indexes for partitions in which the data doesn't change. For example, in a table called Orders, where only the current orders are updated, you can create separate partitions for current orders and completed orders, and only rebuild indexes on the partition that contains the current orders.
- **Performing partial backups.** You can use multiple filegroups to store partitioned tables and indexes. If some partitions use read-only filegroups, you can use partial backups to back up only the primary filegroups and the read/write filegroups; this is more efficient than backing up the entire database.

You can use partitioning to create partitioned tables and partitioned indexes. When you create a partitioned table, you do not create it on a filegroup, as you do a nonpartitioned table. Instead, you create it on a partition scheme, which defines a filegroup or, more usually, a set of filegroups. In turn, a partition scheme is based on a partition function, which defines the boundary values that will be used to divide table data into partitions.

Partition Functions

When creating a partition function, you need to first plan the column in the table that you will use to partition the data. You should then decide which values in that column will be the boundary values. It is common practice in tables that contain a **datetime** or **smalldatetime** column to use this to partition data, because this means you can divide the table based on time intervals. For example, you could partition a table that contains order information by order date. You could then maintain current orders in one partition, and archive older orders into one or more additional partitions.

- When partitioning data, it is common practice to partition using a datetime value
- Use CREATE PARTITION FUNCTION statement
 - RANGE LEFT
 - The default value
 - The maximum value of the partition boundary
 - RANGE RIGHT
 - The minimum value of the partition boundary
- The function can be used on multiple objects

The values that you choose for the partition function will have an effect on the size of each partition. For example, in the **OrderArchive** table, you could choose boundary values that divide data based on yearly intervals. The bigger the gap between the intervals, the bigger each partition will probably be. The number of values that you include in the partition function determines the number of partitions in the table. For example, if you include two boundary values, there will be three partitions in the table. The additional partition is created to store the values outside of the second boundary.

 **Note:** You cannot use the following data types in a partition function: **text**, **ntext**, **image**, **xml**, **timestamp**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, alias data types, and CLR user-defined data types.

To create a partition function, use the **CREATE PARTITION FUNCTION** Transact-SQL statement. You must specify the name of the function, and the data type that the function will use. This should be the same as the data type of the column that you will use as the partitioning key. You must also detail the boundary values, and either **RANGE LEFT** or **RANGE RIGHT** to specify how to handle values in the table that fall exactly on the boundary values:

- **RANGE LEFT** is the default value. This value forms the upper boundary of a partition. For example, if a partition function used a boundary value of midnight on December 31, 2015, any values in the partitioned table date column that were equal to or less than this date and time would be placed into the partition. All values from January 1, 2016 00:00:00 would be stored in the second partition.
- With **RANGE RIGHT**, the value is the lower boundary of the partition. If you specified January 1, 2016 00:00:00, all dates including and later than this date would go into one partition; all dates before this would go into another partition. This produces the same result as the RANGE LEFT example.

 **Note:** The definition of the partition function does not include any objects, columns, or filegroup storage information. This independence means you can reuse the function for as many tables, indexes, or indexed views as you like. This is particularly useful for partitioning dates.

The following code example creates a partition function called **YearlyPartitionFunction** that specifies three boundary values, and will therefore create four partitions:

CREATE PARTITION FUNCTION AS RANGE LEFT Transact-SQL Statement

```
CREATE PARTITION FUNCTION YearlyPartitionFunction (smalldatetime)
AS RANGE LEFT
FOR VALUES ('2013-12-31 00:00', '2014-12-31 00:00', '2015-12-31 00:00');
GO
```

The **YearlyPartitionFunction** in the preceding code example can be applied to any table. After partitioning has been added to a table (you will see this in a later lesson), the datetime column used for partitioning value will determine which partition the row will be stored in:

Partition Number	Minimum Value	Maximum Value
1	Earliest date prior or equal to 2013-12-31 00:00	2013-12-31 00:00
2	2014-01-01 00:00	2014-12-31 00:00
3	2015-01-01 00:00	2015-12-31 00:00
4	2016-01-01 00:00	All dates from 2016-01-01 00:00

If the function used the RANGE RIGHT option, then the maximum values would become the minimum values:

Partition Number	Minimum Value	Maximum Value
1	Earliest date prior or equal to 2013-12-30 00:00	2013-12-30 00:00
2	2013-12-31 00:00	2014-12-30 00:00
3	2014-12-31 00:00	2015-12-30 00:00
4	2015-12-31 00:00	All dates from 2015-12-31 00:00

In this case, RANGE LEFT works better with the dates used, as each partition can then contain data for one year. RANGE RIGHT would work better using the dates in the following code example.

The following code uses RANGE RIGHT to divide rows into annual partitions:

CREATE PARTITION FUNCTION AS RANGE RIGHT Transact-SQL Statement

```
CREATE PARTITION FUNCTION YearlyPartitionFunction (smalldatetime)
AS RANGE RIGHT
FOR VALUES ('2014-01-01 00:00', '2015-01-01 00:00', '2016-01-01 00:00');
GO
```

 **Note:** A table or index can have a maximum of 15,000 partitions in SQL Server.

Partition Schemes

Partition schemes map table or index partitions to filegroups. When planning a partition scheme, think about the filegroups that your partitioned table will use. By using multiple filegroups for your partitioned table, you can separately back up discrete parts of the table by backing up the appropriate filegroup. It is common practice to use one filegroup for each partition, but this is not a requirement; you can use a single filegroup to store all partitioned data, or map some partitions to a single filegroup, and others to separate filegroups.

- Partition schemes map table or index partitions to filegroups:
 - Common practice to use one filegroup per partition
 - Use multiple filegroups for your partitioned tables to back up discrete parts of the table
 - Partitioned data can be stored in a single filegroup, but best for read-only data
 - A partition function must be created before creating the partition scheme

For example, if you plan to store read-only data in a partitioned table, you might place all filegroups that contain read-only data on the same filegroup, so you can manage the data together.

To create a partition scheme, use the **CREATE PARTITION SCHEME** Transact-SQL statement. You must specify a name for the scheme, the partition function that it references, and the filegroups that it will use.

The following code example creates a scheme called **OrdersByYear** that references the function PartitionByYearFunction and uses four filegroups, **Orders1**, **Orders2**, **Orders3**, and **Orders4**:

CREATE PARTITION SCHEME Transact-SQL Statement

```
CREATE PARTITION SCHEME OrdersByYear
AS PARTITION PartitionByYearFunction
TO (Orders1, Orders2, Orders3, Orders4);
GO
```



Note: You must create the partition function using the CREATE PARTITION FUNCTION statement before you create your partition scheme.

Creating a Partitioned Table

After creating your partition function and scheme, you are ready to partition your tables. When you create an object, it includes an ON clause to instruct SQL Server where it should store the object. In common practice, this clause is often omitted and the object is stored on the default filegroup.

To create a partitioned table, you use the CREATE TABLE statement with the ON clause, and specify which partition scheme to use. The partition scheme will then determine which filegroup each row will be stored in. Your table must have an appropriate partition key column for the scheme to be able to partition the data, and it must be the same data type as specified when creating the function.

The following code example creates a table named **Orders**, which will use the **OrdersByYear** scheme:

Creating a Partitioned Table

```
CREATE TABLE Orders
(
    OrderID int IDENTITY(1,1) PRIMARY KEY,
    CustomerID int NOT NULL,
    ShippingAddressID int NOT NULL,
    BillingAddressID int NOT NULL,
    OrderDate smalldatetime NOT NULL
)
ON OrdersByYear(OrderDate);
GO
```

- Use the CREATE TABLE statement to create a partitioned table
- Specify the partition scheme in the ON clause
 - The scheme determines the filegroup for each row
- The scheme uses a partitioning key column to determine how each row is partitioned
 - The data type of this column must match the type declared in the scheme

If the partitioning key is a computed column, this column must be PERSISTED.

Partitioned Indexes

You create a partitioned index in much the same way as a table using the ON clause, but a table and its indexes can be partitioned using different schemes. However, you must partition the clustered index and table in the same way, because the clustered index cannot be stored separately from the table. If a table and all its indexes are identically partitioned by using the same partition scheme, then they are considered to be **aligned**. When storage is aligned, both the rows in a table and the indexes that depend on these rows will be stored in the same filegroup. Therefore, if a single partition is backed up or restored, both the data and indexes are kept together. An index that is partitioned differently to its dependent table is considered **nonaligned**.

- Use CREATE INDEX or CREATE TABLE to create an index:
- Partition the index using the ON clause to specify schema
- Nonclustered indexes can use a different partition scheme to the table
- Clustered index must use the same schema as the table
- The column used for partition does not have to be part of the index
- Indexes using same schema as table are **aligned**
- Rebuild or reorganize the entire index, or one partition

An index is partitioned by specifying a partition scheme in the ON clause.

The following example creates a nonclustered index on the OrderID column of the Orders table:

Create a Partitioned Nonclustered Index

```
CREATE NONCLUSTERED INDEX ixOrderID
ON dbo.Orders (OrderID) ON OrdersByYear(OrderDate);
```

Notice that, when you partition an index, you are not limited to using the columns in the index when specifying the partitioning key. SQL Server includes the partitioning key in the definition of the index, which means you can partition the index using the same schema as the table.

The following code creates the Orders table with a partitioned clustered index on the OrderID and OrderDate column of the Orders table:

Create a Table with a Partitioned Clustered Index

```
CREATE TABLE dbo.Orders
(
    OrderID int IDENTITY(1,1),
    OrderDate datetime NOT NULL,
    CONSTRAINT PK_Orders PRIMARY KEY CLUSTERED (OrderDate, OrderID)
)
ON OrdersByYear(OrderDate);
```

When an index is partitioned, you can rebuild or reorganize the entire index, or a single partition of an index. The **sys.dm_db_index_physical_stats** dynamic management view (DMV) provides fragmentation information for each partition, so you can see which partitions are most heavily fragmented. You can then create a targeted defragmentation strategy based on this data.

The following code uses the **sys.dm_db_index_physical_stats** DMV to show fragmentation in each partition of a table:

Show Fragmentation by Partition Using **sys.dm_db_index_physical_stats**

```
SELECT i.name, s.index_type_desc, s.partition_number, avg_fragmentation_in_percent
FROM sys.dm_db_index_physical_stats
(DB_ID(N'TSQL'), OBJECT_ID(N'dbo.Orders'), NULL, NULL, NULL) AS s
INNER JOIN sys.indexes AS i ON s.object_id = i.object_id AND s.index_id = i.index_id;
```

SWITCH, MERGE and SPLIT Operations

You can manipulate the partitions in a partitioned table by performing SWITCH, MERGE and SPLIT operations, and by using dedicated partitioning functions and catalog views.

Switching Partitions

One of the major benefits of partitioned tables is the ability to switch individual partitions in and out of a partitioned table. By using switching, you can archive data quickly and with minimal impact on other database operations. This is because, if it is configured correctly, switching usually only involves swapping the metadata of two partitions in different tables, not the actual data. Consequently, the operation has minimal effect on performance. You can switch partitions between partitioned tables, or you can switch a partition from a partitioned table to a nonpartitioned table.

Consider the following points when planning partition switching:

- Both the source partition (or table) and the destination partition (or table) must be in the same filegroup, so you need to take account of this when planning filegroups for a database.
- The target partition (or table) must be empty; you cannot perform a SWITCH operation by using two populated partitions.
- The two partitions or tables involved must have the same schema (columns, data types, and so on). The rows that the partitions contain must also fall within exactly the same range of values for the partitioning column; this ensures that you cannot switch rows with inappropriate values into a partitioned table. You should use CHECK constraints to ensure that the partitioning column values are valid for the partition being switched. For example, for a table that is partitioned by a date value, you could create a CHECK constraint on the table that you are switching; this then checks that all values fall between two specified dates.

Splitting Partitions

Because you need to maintain an empty partition to switch partitions, it is usually necessary to split an existing partition to create a new empty partition that you can then use to switch data. To split a partition, you first need to alter the partition scheme to specify the filegroup that the new partition will use (this assumes that your solution maps partitions one-to-one with filegroups). When you alter the scheme, you specify this filegroup as the next used filegroup, which means that it will automatically be used for the new partition that you create when you perform the split operation.

The following code example adds the next used filegroup **NewFilegroup** to the **OrderArchiveScheme** partition scheme:

ALTER PARTITION SCHEME Transact-SQL Statement

```
ALTER PARTITION SCHEME OrderArchiveScheme NEXT USED NewFilegroup;
GO
```

You can then alter the partition function to split the range and create a new partition.

- Use the following functions to manage partitions:
 - **SPLIT**
 - Add a new boundary to split a single partition into two partitions
 - **SWITCH**
 - Switch a partition between partitioned tables, or between a partitioned table and a nonpartitioned table
 - **MERGE**
 - Remove a boundary to merge two partitions into a single partition

The following code example adds a new partition by splitting the range:

ALTER PARTITION FUNCTION Transact-SQL Statement

```
ALTER PARTITION FUNCTION OrderArchiveFunction () SPLIT RANGE ('2012-07-01 00:00')
```

You can now switch the empty partition as required. To do this, you need the partition number, which you can get by using the \$PARTITION function, and specifying the value for which you want to identify the partition.

The following code example switches a partition from the Orders table to the OrderArchive table:

ALTER TABLE Transact-SQL Statement

```
DECLARE @p int = $PARTITION.OrderArchiveFunction('2012-04-01 00:00');
ALTER TABLE Orders
SWITCH TO OrderArchive PARTITION @p
GO
```

Merging Partitions

Merging partitions does the opposite of splitting a partition, because it removes a range boundary instead of adding one.

The following code example merges two partitions:

ALTER TABLE Transact-SQL Statement

```
ALTER PARTITION FUNCTION OrderArchiveFunction
MERGE RANGE ('2011-04-01 00:00');
GO
```

Designing Partition Strategies for Common Scenarios

You can use SWITCH, MERGE, and SPLIT to implement a sliding window strategy for archiving or purging data. For example, if you have a table that stores current orders, you could periodically SWITCH data that is older than three months out of this table and into a staging table, and then archive it. Managing old data by using a sliding window is much more efficient than extracting data from a table, because the SWITCH operation does not require you to move the data.

A table that participates in a sliding window strategy typically includes:

- **A partition function that uses a datetime column.** To implement a sliding window, you should use a datetime data type.
- **Partitions that map to the appropriate time period.** For example, if you want to SWITCH out one month's data at a time, each partition should contain only the data for a single month. You specify the time periods by defining the boundary values in the partition function.
- **Empty partitions.** Performing MERGE and SPLIT operations on empty partitions maintains the number of partitions in the table, and makes the table easier to manage.

- Use **SWITCH**, **MERGE**, and **SPLIT** to implement a sliding window strategy to archive or purge data
- Switch old data to a staging table, and then archive it
- Efficient—as metadata is updated, data is not moved
- Considerations:
 - Choose partition boundary carefully, as SQL Server performs explicit rounding of time in datetime values
 - Create a CHECK constraint on the staging table to which you will switch the partition so it contains the same dates; NULLs are not allowed

Sliding Windows Strategies

The way that you implement a sliding window depends on whether you use RANGE LEFT or RANGE RIGHT in the partition function. If you use RANGE LEFT to create the partition function, you can create and maintain a partitioned table as described in the following example:

1. Create a partitioned table with four partitions, each of which represents a period of one month. Partition 1 contains the oldest data, partition 2 contains the current data, partition 3 is empty, and partition 4 is empty. The table looks like this:

Partition 1: Oldest Data	Partition 2: Current Data	Partition 3: Empty	Partition 4: Empty
-----------------------------	------------------------------	-----------------------	-----------------------

2. Switch out partition 1 to a staging table for purging or archiving. The table now looks like this:

Partition 1: Empty	Partition 2: Current Data	Partition 3: Empty	Partition 4: Empty
-----------------------	------------------------------	-----------------------	-----------------------

3. Merge the now empty partition 1 with partition 2. This partition now contains the oldest data. The table now has three partitions, one populated and two empty. The empty middle partition (which was partition 3) will be used to load the current data. The table now looks like this:

Partition 1: Oldest Data	Partition 2: Empty: load current data	Partition 3: Empty
-----------------------------	---	-----------------------

4. Split the other empty partition to return the table to the same state as it was in step 1.

If you use RANGE RIGHT to create the partition function, you can create and maintain a partitioned table as described in the following example:

1. Create a partitioned table with four partitions, each of which represents a period of one month. Partition 1 is empty, partition 2 contains the oldest data, partition 3 contains the current data, and partition 4 is empty. The table looks like this:

Partition 1: Empty	Partition 2: Oldest Data	Partition 3: Current Data	Partition 4: Empty
-----------------------	-----------------------------	------------------------------	-----------------------

2. Switch out partition 2 to a staging table. The table now looks like this:

Partition 1: Empty	Partition 2: Empty	Partition 3: Current Data	Partition 4: Empty
-----------------------	-----------------------	------------------------------	-----------------------

3. Merge the empty partitions 1 and 2. The table now has three partitions, and looks like this:

Partition 1: Empty	Partition 2: Oldest Data	Partition 3: Empty
-----------------------	-----------------------------	-----------------------

4. Split partition 3 to return the table to the same state as it was in step 1.

Considerations for Implementing a Sliding Window

When planning a sliding window strategy, consider the following points:

- **Partition boundary values with RANGE LEFT.** When partitioning on a column that uses the **datetime** data type, you should choose the partition boundary value that you specify with RANGE LEFT carefully. SQL Server performs explicit rounding of times in **datetime** values that can have unexpected consequences. For example, if you create a partition function with a RANGE LEFT boundary value of **2012-10-30 23:59:59.999**, SQL Server will round this up to **2012-10-31 00:00:00.000**; as a result, rows with the value of midnight will be added to the left partition instead of the right. This could lead to inconsistencies because some rows for a particular date might be in a different partition to the other rows with the same date. To avoid SQL Server performing rounding on times in this way, specify the boundary value as **2012-10-30 23:59:59.997** instead of **2012-10-30 23:59:59.999**; this will ensure that rows are added to partitions as expected. For the **datetime2** and **datetimeoffset** data types, you can specify a boundary of **2012-10-30 23:59:59.999** without experiencing this problem. If you use RANGE RIGHT in the partition function, specifying a time value of **00:00:00:000** will ensure that all rows for a single date are in the same partition, regardless of the data type that you use.
- **CHECK constraint.** You must create a check constraint on the staging table to which you will switch the partition containing the old data. The check constraint should ensure that both partitions contain dates for exactly the same period, and that NULL values are not allowed.

The code example below adds a check constraint to the Orders_Staging table:

ALTER TABLE ...WITH CHECK Transact-SQL Statement

```
ALTER TABLE Orders_Staging
WITH CHECK ADD CONSTRAINT CheckDates
CHECK (OrderDate >= '2010-07-01' AND OrderDate < '2010-10-01' AND OrderDate IS NOT NULL);
GO
```

Demonstration: Creating a Partitioned Table

In this demonstration, you will see how to partition data.

Demonstration Steps

Creating a Partitioned Table

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **D:\Demofiles\Mod03\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**, and then if prompted with the question **Do you want to continue with this operation?** type **Y**, then press Enter.
4. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows® authentication.
5. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod03\Demo** folder.
6. In Solution Explorer, open the **1 - Partitioning.sql** script file.
7. Select and execute the query under **Step 1** to use the master database.
8. Select and execute the query under **Step 2** to create four filegroups, and add a file to each filegroup.

9. Select and execute the query under **Step 3** to switch to the **AdventureWorks** database.
10. Select and execute the query under **Step 4** to create the partition function.
11. Select and execute the query under **Step 5** to create the **OrdersByYear** partition scheme.
12. Select and execute the query under **Step 6** to create the **Sales.SalesOrderHeader_Partitioned** table.
13. Select and execute the query under **Step 7** to copy data into the **Sales.SalesOrderHeader_Partitioned** table.
14. Select and execute the query under **Step 8** to check the rows counts within each of the partitions.
15. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Question	
What is the maximum number of partitions you can add to a table or index?	
Select the correct answer.	
	4
	256
	1,000
	15,000
	256,000

Lesson 2

Compressing Data

SQL Server includes the ability to compress data in SQL Server databases. Compression reduces the space required to store data and can improve performance for workloads that are I/O intensive. This lesson describes the options for using SQL Server compression and its benefits. It also describes the considerations for planning data compression.

 **Note:** In versions of SQL Server before SQL Server 2016 Service Pack 1, compression was only available in Enterprise edition. In SQL Server 2016 Service Pack 1 and later, compression is available in all editions of SQL Server.

Lesson Objectives

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

- Describe the benefits of data compression.
- Add page compression to your databases.
- Use row compression on your tables.
- Add Unicode compression to your databases.
- Understand the considerations for compressing data.

Why Compress Data?

SQL Server data compression can help you to achieve savings in storage space. For workloads that require a lot of disk I/O activity, using compression can improve performance. This performance boost occurs because compressed data requires fewer data pages for storage, so queries that access compressed data require fewer pages to be retrieved.

You can use SQL Server compression to compress the following SQL Server objects:

- Tables
- Nonclustered indexes
- Indexed views
- Individual partitions in a partitioned table or index—each partition can be set to PAGE, ROW or NONE
- Spatial indexes

- Data compression:
 - Helps save storage space
 - Improves performance for workloads requiring heavy disk I/O activity, as queries accessing compressed data retrieve fewer pages
- Can be added to the following objects:
 - Tables stored as heaps
 - Tables stored as clustered indexes
 - Nonclustered indexes
 - Indexed views
 - Individual partitions
 - Spatial indexes

In SQL Server, you can implement compression in two ways: page compression and row compression. You can also implement Unicode compression for the **nchar(n)** and **nvarchar(n)** data types.



Note: You can see the compression state of the partitions in a partitioned table by querying the **data_compression** column of the **sys.partitions** catalog view.

Page Compression

Page compression takes advantage of data redundancy to reclaim storage space. Each compressed page compression includes a structure called the compression information (CI) structure below the page header. The CI structure is used to store compression metadata.

Page compression compresses data in three ways:

1. **Row compression.** When you implement page compression, SQL Server automatically implements row compression; in other words, page compression incorporates row compression.
2. **Prefix compression.** SQL Server scans each compressed column to identify values that have a common prefix. It then records the prefixes in the CI structure and assigns an identifier for each prefix—which it then uses in each column to replace the shared prefixes. Because an identifier is usually much smaller than the prefix that it replaces, SQL Server can potentially reclaim a considerable amount of space. For example, imagine a set of parts in a product table that all have an identifier that begins TFG00, followed by a number. If the Products table contains a large number of these products, prefix compression would eliminate the redundant TFG00 values from the column, and replace them with a smaller alias.
3. **Dictionary compression.** Dictionary compression works in a similar way to prefix compression, but instead of just identifying prefixes, dictionary compression identifies entire repeated values, and replaces them with an identifier that is stored in the CI structure. For example, in the products table, there is a column called Color that contains values such as Blue, Red, and Green that are repeated extensively throughout the column. Dictionary compression would replace each color value with an identifier, and store each color value in the CI structure, along with its corresponding identifier.

- Page compression:
 - Adds a compression information (CI) structure below the page header on each compressed page
 - Takes advantage of data redundancy to claim space
- Compresses data in three ways:
 - Row compression: page compression incorporates row compression
 - Prefix compression: replaces common prefix values with smaller identifier, which is stored in the CI structure
 - Dictionary compression: similar to prefix, but replaces repeated values with smaller identifier, also stored in the CI structure

SQL Server processes these three compression operations in the order that they are shown in the previous bullet list.

Row Compression

Row compression saves space by changing the way it stores fixed length data types. Instead of storing them as fixed length types, row compression stores them in variable length format. For example, the integer data type normally takes up four bytes of space. If you had a column that used the integer data type, the amount of space that each row has in the column would vary, depending on the values in the rows. A value of six would only consume a single byte, whereas a value of 6,000 would consume two bytes. Row compression only works with certain types of data; it does not affect variable length data types, or other data types including: **xml**, **image**, **text**, and **ntext**.

When you implement row compression for a table, SQL Server adds an extra four bits to each compressed column to store the length of the data type. However, this small increase in size is normally outweighed by the space saved. For NULL values, the four bits is the only space that they consume.

- Row compression:
 - Saves space by storing fixed length data types as variable length, such as integer types
 - Does not work with variable length data types, XML, image, text and ntext
 - Adds an extra four bits to each compressed column to store the length of the data type
 - For NULL values, the four bits is the only space consumed

Unicode Compression

SQL Server Unicode compression uses the Standard Compression Scheme for Unicode (SCSU) algorithm to extend the compression capabilities to include the Unicode data types **nchar(n)** and **nvarchar(n)**. When you implement row or page compression on an object, Unicode columns are automatically compressed by using SCSU. Note that **nvarchar(MAX)** columns cannot be compressed with row compression, but can benefit from page compression.

Unicode data types use more bytes to store the equivalent non-Unicode values. SQL Server uses the UCS-2 Unicode encoding scheme, which uses two bytes to store each character. Non-Unicode data types use only one byte per character. The SCSU algorithm essentially stores Unicode values as non-Unicode values and converts them back to Unicode as required. This can yield significant storage savings, with compression for English language Unicode data types yielding 50 percent savings.

- Unicode compression uses the Standard Compression Scheme for Unicode (SCSU) algorithm
- **nchar(n)** and **nvarchar(n)** data types automatically compressed with SCSU
- **nvarchar(MAX)** can be compressed with page but not row compression
- Compression ratio varies between different languages: English yields 50% savings, Japanese yields 15%



Note: The compression ratio for Unicode compression varies between different languages, particularly for languages whose alphabets contain significantly more characters. For example, Unicode compression for the Japanese language yields just 15 percent savings.

Considerations for Compression

The benefits of using SQL Server compression include the ability to reclaim storage space and improved performance. However, compressing and uncompressing data requires a significant amount of server resources, particularly processor resources. Consequently, you should plan compression to ensure that the benefits of compressing data will outweigh the costs.

- Plan compression to save storage space:
 - Consider the characteristic of the data to decide whether a table or index is a good candidate for compression
 - Consider compressing tables with many fixed data type columns, or with large amounts of redundant data
- Plan compression to improve performance:
 - Compression can reduce disk I/O and increase the amount of data that SQL Server can hold in memory
 - Compression increases CPU utilization
 - Balance the performance benefit against the increased CPU utilization

Planning Compression to Reclaim Storage Space

The amount of storage space that you can reclaim by using compression depends on the type of data that you are compressing. Consider the following points when planning data compression:

- If your data includes a large number of fixed length data types, you can potentially benefit from row compression. For the greatest benefit, a large number of the values should consume less space than the data types allow in total. For example, the **smallint** data type consumes 2 bytes of space. If a smallint column contains values that are less than 256, you can save a whole byte of data for each value. However, if the majority of the values are greater than this, the benefit of compressing the data is less, because the overall percentage of space saved is lower.
- If your data includes a large amount of redundant, repeating data, you might be able to benefit from page compression. This applies to repeating prefixes, in addition to entire words or values.
- Whether or not you will benefit from Unicode compression depends on the language that your data is written in.

You can use the stored procedure **sp_estimate_data_compression_savings** to obtain an estimation of the savings that you could make by compressing a table. When you execute **sp_estimate_data_compression_savings**, you supply the schema name, the table name, the index id if this is included in the calculation, the partition id if the table is partitioned, and the type of compression (ROW, PAGE, or NONE). **sp_estimate_data_compression_savings** takes a representative sample of the table data and places it in **tempdb**, where it is compressed, and then supplies a result set that displays the potential savings that you could make by compressing the table.

The following code estimates the potential space savings of implementing row compression in the Internet.Orders table:

sp_estimate_data_compression_savings

```
USE Sales;
GO
EXEC sp_estimate_data_compression_savings 'Internet', 'Orders', NULL, NULL, 'ROW';
GO
```

Planning Compression to Improve Performance

Because compressed data uses fewer data pages, reading compressed data requires fewer page reads, which in turn requires reduced disk I/O, the result of which can be improved performance. Compressed data is stored on disk in a compressed state, and it remains in a compressed state when the data pages are loaded into memory. Data is only uncompressed when it is required, for example, for join operations, or when an application reads the data. As a result, SQL Server can fit more pages into memory than if the data were uncompressed, which can potentially boost performance further. The benefits of reduced disk

I/O and more efficient in-memory page storage are greater for workloads that scan large amounts of data, rather than for queries that return just a small subset of data.

To decide whether to implement compression, you must balance the performance improvements against the cost, in terms of CPU resources, of compressing and uncompressing the data. For row compression, this cost is typically a seven to 10 percent increase in CPU utilization. For page compression, this figure is usually higher.

Two factors that can help you to assess the value of implementing compression for a table or index are:

1. The frequency of data change operations relative to other operations. The lower the percentage of data change operations, such as updates, the greater the benefit of compression. Updates typically require access to only a small part of the data, and so do not involve accessing a large number of data pages.
2. The proportion of operations that involve a scan. The higher this value, the greater the benefit of compression. Scans involve a large number of data pages, so you can improve performance considerably if a significant percentage of the workload involves scans.

You can use the **sys.dm_db_index_operational_stats** dynamic management view (DMV) to obtain the information required to assess the frequency of data change operations and the proportion of operations that involve a scan.

Demonstration: Compressing Data

In this demonstration, you will see how to compress data.

Demonstration Steps

Compressing Data

1. In Server Management Studio, in Solution Explorer, open the **2 - Compressing Data.sql** script file.
2. Select and execute the query under **Step 1** to use the **AdventureWorks** database.
3. Select and execute the query under **Step 2** to run the **sp_estimate_data_compression_savings** procedure against the **Sales.SalesOrderDetail** table.
4. Select and execute the query under **Step 3** to add row compression to the **Sales.SalesOrderDetail** table.
5. Select and execute the code under **Step 4** to run the **sp_estimate_data_compression_savings** procedure against the **Sales.SalesOrderDetail** table to see if the table can be further compressed.
6. Select and execute the query under **Step 5** to rebuild indexes 1 and 3.
7. Select and execute the query under **Step 6** to run **sp_estimate_data_compression_savings** procedure against the **Sales.SalesOrderDetail** table to show how the size of the table has been reduced.
8. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Question
You have a Customers table with the following columns: Title, FirstName, MiddleInitial, LastName, Address1, Address2, City, PostalCode, Telephone, and Email. Which of the following options will give the best reduction in storage?
Select the correct answer.
<input type="checkbox"/> Add ROW compression to the table.
<input type="checkbox"/> Add PAGE compression to the table.
<input type="checkbox"/> Add Unicode compression to the table.
<input type="checkbox"/> Create a nonclustered index on the FirstName and LastName columns.
<input type="checkbox"/> None of the above.

Lesson 3

Temporal Tables

Most developers, at some point in their careers, have faced the problem of capturing and storing changed data, including what was changed, and when it was changed. In addition to the Slowing Changing Dimension (SCD) component found in data warehousing, it is common to add triggers or custom code to extract the changes and store this for future reference. The introduction of temporal tables means that SQL Server can capture data change information automatically. These tables are also known as system-versioned tables.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the benefits of using temporal tables in your database.
- Create temporal tables.
- Add system-versioning to an existing table.
- Understand considerations, including limitations, of using temporal tables.
- Use system-versioning with memory optimized tables.

What are System-Versioned Temporal Tables?

Temporal tables solve the issue of capturing and storing changes to the data. Developers often need to capture changes for auditing or reporting purposes. The slowing changing dimension (SCD) component in a data warehouse is a common way to capture changes from the OLTP system, so that data can be viewed and analyzed over periods of time. However, in an OLTP system, a simpler approach is often more appropriate. Temporal tables solve this issue, and can also be queried within the database.

With the temporal table feature, you can record all changes to your data. You create a system-versioned table either by creating a new table, or modifying an existing table. When a new table is created, a pair of tables are created—one for the current data, and one for the historical data. Two **datetime2** columns are added to both the current and historical tables to store the valid date range of the data: **SysStartTime** and **SysEndTime**. The current row will have a **SysEndTime** value of **9999-12-31**, and all records inserted within a single transaction will have the same UTC time. When a row is updated, the data prior to the update is copied to the historical table, and the **SysEndTime** column is set to the date that the data is changed. The row in the current table is then updated to reflect the change.

For a relationship to be established with the historical table, the current table must have a primary key. This also means you can indirectly query the historical table to see the full history for any given record. The historical table can be named at the time of creation, or SQL Server will give it a default name.

- System-versioned tables store a full history of data changes:
 - Current and historical tables operate as a pair
 - SysStartTime and SysEndTime columns store the valid dates of the data
 - Current table must have a primary key
 - All versioning is automatic, no developer intervention required
 - Can be added to existing tables
 - Historical table can be named, or take system name

Creating a Temporal Table

To create a new system-versioned table, use the standard CREATE TABLE code with two datetime2 start and end time columns, declare these columns as the dates for the PERIOD FOR SYSTEM_TIME, and then specify system-versioning ON.

Create a new table and set the SYSTEM_VERSIONING feature ON.

Create Temporal Table

```
CREATE TABLE dbo.Employee
(
    EmployeeID int NOT NULL PRIMARY KEY
    CLUSTERED,
    ManagerID int NULL,
    FirstName varchar(50) NOT NULL,
    LastName varchar(50) NOT NULL,
    SysStartTime datetime2 GENERATED ALWAYS AS ROW START NOT NULL,
    SysEndTime datetime2 GENERATED ALWAYS AS ROW END NOT NULL,
    PERIOD FOR SYSTEM_TIME (SysStartTime, SysEndTime)
)
WITH (SYSTEM_VERSIONING = ON (HISTORY_TABLE = dbo.EmployeeHistory));
```

- To create a temporal table
 - SysStartTime column—or another datetime2 column
 - SysEndTime column—or another datetime2 column
 - SYSTEM_VERSIONING = ON
 - Optionally name the HISTORY_TABLE

You must include the **SysStartTime** and **SysEndTime** columns, and the **PERIOD FOR SYSTEM_TIME** which references these columns. You can change the name of these columns and change the references to them in the PERIOD FOR SYSTEM_TIME parameters, as shown in the following code:

The Manager table in the following example has the date columns named as **DateFrom** and **DateTo**—these names are also used in the history table:

The Manager table in the following example has the date columns named as **DateFrom** and **DateTo**—these names are also used in the history table:

Change the Names of the Start and End System Columns

```
CREATE TABLE dbo.Manager
(
    ManagerID int NOT NULL PRIMARY KEY CLUSTERED,
    FirstName varchar(50) NOT NULL,
    LastName varchar(50) NOT NULL,
    DateFrom datetime2 GENERATED ALWAYS AS ROW START NOT NULL,
    DateTo datetime2 GENERATED ALWAYS AS ROW END NOT NULL,
    PERIOD FOR SYSTEM_TIME (DateFrom, DateTo)
)
WITH (SYSTEM_VERSIONING = ON (HISTORY_TABLE = dbo.ManagerHistory));
```

If you want SQL Server to name the historical table, run the preceding code excluding the (HISTORY_TABLE = dbo.EmployeeHistory) clause. Furthermore, if you include the **HIDDEN** keyword when specifying the start and end time columns, they don't appear in the results of a SELECT * FROM statement. However, you can specify the start and end column names in the select list to include them.

Adding System-Versioning to an Existing Table

An existing table can be converted to a system-versioned table by:

- Adding the start and end datetime2 columns.
- Setting the system-versioning flag on.

The following code adds the system datetime2 columns to the Sales table. After these are created, the code alters the Sales table so that

SYSTEM_VERSIONING is ON and data changes are stored. In this example, the history table has been named:

- Convert an existing table to a system-versioned table:
 - Add the start and end date columns
 - Set the system-versioning flag on

Make an Existing Table System-Versioned

```
ALTER TABLE dbo.Sales
ADD
SysStartTime datetime2(0) GENERATED ALWAYS AS ROW START CONSTRAINT DF_SysStartTime
DEFAULT SYSUTCDATETIME(),
SysEndTime datetime2(0) GENERATED ALWAYS AS ROW END CONSTRAINT DF_SysEndTime DEFAULT
CONVERT(datetime2 (0), '9999-12-31 23:59:59'),
PERIOD FOR SYSTEM_TIME (SysStartTime, SysEndTime);
GO

ALTER TABLE dbo.Sales
SET (SYSTEM_VERSIONING = ON (HISTORY_TABLE = dbo.SalesHistory));
```

Temporal Table Considerations

There are a number of points that you should consider before using a system-versioned table:

- The system date columns SysStartTime and SysEndTime must use the datetime2 data type.
- The current table must have a primary key, but the historical table cannot use constraints.
- To name the history table, you must specify the schema name as well as the table name.
- By default, the compression of the history table is PAGE compressed.
- The history table must reside in the same database as the current table.
- System-versioned tables are not compatible with FILETABLE or FILESTREAM features because SQL Server cannot track changes that happen outside of itself.
- Columns with a BLOB data type, such as varchar(max) or image, can result in high storage requirements because the history table will store the history values as the same type.
- INSERT and UPDATE statements cannot reference the SysStartTime or SysEndTime columns.
- You cannot modify data in the history table directly.

- Considerations when using system-versioned tables:
 - SysStartTime and SysEndTime must be datetime2
 - Current table must have a primary key
 - To name the history table, must include schema name
 - By default the history table is PAGE compressed
 - History table must reside in same database
 - Does not support FILETABLE or FILESTREAM
 - Blob data types can result in high storage requirements
 - Data in the historical table cannot be directly modified
 - Current table cannot be truncated when SYSTEM_VERSIONING is ON

- You cannot truncate a system-versioned table. Turn SYSTEM_VERSIONING OFF to truncate the table.
- Merge replication is not supported.

For a full list of considerations and limitations, see Microsoft Docs:

Temporal Table Considerations and Limitations

<http://aka.ms/vlehj6>

System-Versioned Memory-Optimized Tables

In-Memory OLTP was introduced in SQL Server 2014 so that tables could exist in memory, giving optimal performance by avoiding concurrency, and reading data from memory rather than disk. These high performance tables are known as memory-optimized tables.

All changes to a memory-optimized table are stored in a transaction log that resides on disk. This guarantees that the data is secure in the event of a service restart. System-versioning can be added to a memory-optimized table, but rather than bloating the server memory with the data changes stored in the history table, and possibly exceeding the RAM limit, changed data is stored on disk. SQL Server also creates an internal memory-optimized staging table that sits between the current memory-optimized table and the disk-based history table. This is covered in more detail later in this topic.

Considerations for system-versioned memory-optimized tables:

- Durability of current table must be SCHEMA_AND_DATA
- Primary key on current table must be nonclustered index
- Internal staging table stores recent changes for fast querying. Flushed to disk-based table on regular interval
- Staging table includes 8 bytes bigint column for uniqueness, so max row size reduced to 8052 bytes
- Include HIDDEN keyword with the period columns to exclude them from SELECT * queries
- Use FOR SYSTEM_TIME to query historical data, including a subclause: AS OF, FROM TO, BETWEEN AND, CONTAINED IN, or ALL

Create a Memory-Optimized Filegroup

To create a memory-optimized table with system-versioning, you must first ensure the database has a filegroup allocated for memory-optimized data. To check for a filegroup, open **SQL Server Management Studio**, and connect to the server. Right-click the database in **Object Explorer** and click **Properties**. Look in the **Filegroups** tab to see if a file exists in the **MEMORY OPTIMIZED DATA** box. You can only have one memory-optimized filegroup per database. If you need to add a filegroup, click **Add**, and enter the name of the filegroup, then click **OK**. You can also use the following code example to create a new memory-optimized filegroup:

Add a filegroup to your database to contain your memory-optimized tables.

Add Filegroup for Memory-Optimized Data

```
ALTER DATABASE AdventureWorks
ADD FILEGROUP MemoryOptimized CONTAINS MEMORY_OPTIMIZED_DATA
ALTER DATABASE AdventureWorks
ADD FILE (name='AdventureWorks_mod', filename='C:\Program Files\Microsoft SQL
Server\MSSQL13.MSSQLSERVER\MSSQL\DATA\AdventureWorks_mod') TO FILEGROUP MemoryOptimized

ALTER DATABASE AdventureWorks
SET MEMORY_OPTIMIZED_ELEVATE_TO_SNAPSHOT = ON
```

Create a System-Versioned Memory-Optimized Table

Now that you have a filegroup for your memory-optimized data, you can create a memory-optimized table with system-versioning. However, there are a few considerations you need to be aware of:

- The durability of the current temporal table must be SCHEMA_AND_DATA.
- The primary key must be a nonclustered index; a clustered index is not compatible.
- If SYSTEM_VERSIONING is changed from ON to OFF, the data in the staging buffer is moved to disk.
- The staging table uses the same schema and the current table, but also includes a **bigint** column to guarantee that the rows moved to the internal buffer history are unique. This bigint column adds 8 bytes, thereby reducing the maximum row size to 8052 bytes.
- Staging tables are not visible in Object Explorer, but you can use the **sys.internal_tables** view to acquire information about these objects.
- If you include the **HIDDEN** keyword when specifying the start and end time columns, they don't appear in the results of a **SELECT * FROM** statement. However, you can specify the start and end column names in the select list to include them.

The following code creates a memory-optimized table with system-versioning enabled. The start and end columns are HIDDEN:

Create a System-Versioned Memory-Optimized Table

```
CREATE TABLE dbo.Employee
(
    EmployeeID int IDENTITY(1,1) NOT NULL PRIMARY KEY NONCLUSTERED,
    Title varchar(4) NOT NULL,
    FirstName nvarchar(25) NOT NULL,
    LastName nvarchar(25) NOT NULL,
    Email nvarchar(50) NOT NULL,
    Telephone nvarchar(15) NULL,
    DateOfBirth date NULL,
    StartTime datetime2(0) GENERATED ALWAYS AS ROW START HIDDEN
        CONSTRAINT DF_EmpStartTime DEFAULT SYSUTCDATETIME() NOT NULL,
    EndTime datetime2(0) GENERATED ALWAYS AS ROW END HIDDEN
        CONSTRAINT DF_EmpEndTime DEFAULT CONVERT(datetime2 (0), '9999-12-31 23:59:59')
    NOT NULL,
    PERIOD FOR SYSTEM_TIME (StartTime, EndTime)
)
WITH
(
    MEMORY_OPTIMIZED = ON,
    DURABILITY = SCHEMA_AND_DATA,
    SYSTEM_VERSIONING = ON (HISTORY_TABLE = dbo.EmployeeHistory)
);
```

Working with the System-Versioned Memory-Optimized Table

Query execution performance will still be fast, despite the history table residing on disk because of the internal, auto-generated memory-optimized staging table that stores recent history. This means you can run queries from natively compiled code. SQL Server regularly moves data from this staging table to the disk-based history table using an asynchronous data flush task. The data flush aims to keep the internal memory buffers at less than 10 percent of the memory consumption of the parent object. If the workload is light, this runs every minute, but may be as frequent as every five seconds for a heavy workload.



Note: You can force a data flush to run by executing `sp_xtp_flush_temporal_history` command, and passing in the name of the schema and table:

`sys.sp_xtp_flush_temporal_history @schema_name, @object_name.`

If you execute a SELECT query against a temporal table, all rows returned will be current data. The SELECT clause is exactly the same query as you would use with a standard user table. To query data in your temporal table for a given point in time, include the **FOR SYSTEM_TIME** clause with one of the five subclauses for setting the datetime boundaries:

1. **AS OF** <date_time> accepts a single datetime parameter and returns the state of the data for the specified point in time.
2. **FROM** <start_date_time> **TO** <end_date_time> returns all current and historical rows that were active during the timespan, regardless of whether they were active before or after those times. The results will include rows that were active precisely on the lower boundary defined by the FROM date; however, it excludes rows that became inactive on the upper boundary defined by the TO date.
3. **BETWEEN** <start_date_time> **AND** <end_date_time> is identical to the FROM TO subclause, except that it includes rows that were active on the upper time boundary.
4. **CONTAINED IN** (<start_date_time>, <end_date_time>) returns the values for all rows that were opened and closed within the specified timespan. Rows that became active exactly on the lower boundary, or became inactive exactly on the upper boundary, are included.
5. **ALL** returns all data from the current and historical tables with no restrictions.

The AS OF subclause returns the data at a given point in time. The following code uses the datetime2 format to return all employees on a specific date—in this case June 1, 2015—and active at 09:00.

Querying a Temporal Table Using the FOR SYSTEM_TIME AS OF Clause

```
SELECT * FROM dbo.Employee
FOR SYSTEM_TIME AS OF '2015-06-01 09:00:00';
```



Best Practice: If you want to return just historical data, use the CONTAINED IN subclause for the best performance, as this only uses the history table for querying.

The FOR SYSTEM_TIME clause can be used to query both disk-based and memory-optimized temporal tables. For more detailed information on querying temporal tables, see Microsoft Docs:



Querying Data in a System-Versioned Temporal Table

<http://aka.ms/y1w3oq>

Demonstration: Adding System-Versioning to an Existing Table

In this demonstration, you will see how:

- System-versioning can be added to an existing table.
- Changes to the data are stored in the temporal table.

Demonstration Steps

Adding System-Versioning to an Existing Table

1. In SQL Server Management Studio, in Solution Explorer, open **the 3 - Temporal Tables.sql** script file.
2. Select and execute the query under **Step 1** to use the AdventureWorks database.
3. Select and execute the query under **Step 2 Add the two date range columns**, to add the two columns, **StartDate** and **EndDate**, to the **Person.Person** table.
4. Select and execute the query under **Step 2 Enable system-versioning**, to alter the table and add system-versioning.
5. In Object Explorer, expand **Databases**, expand **AdventureWorks2016**, right-click **Tables**, and click then **Refresh**.
6. In the list of tables and point out the **Person.Person** table. The name includes **(System-Versioned)**.
7. Expand the **Person.Person (System-Versioned)** table node to display the history table. Point out the name of the table included **(History)**.
8. Expand the **Person.Person_History (History)** node, and then expand **Columns**. Point out that the column names are identical to the current table.
9. Select and execute the query under **Step 4** to update the row in the **Person.Person** table for BusinessEntityID 1704.
10. Select and execute the query under **Step 5** to show the history of changes for BusinessEntityID 1704.
11. Close SQL Server Management Studio without saving anything.

Check Your Knowledge

Question
Which of the following statements is incorrect?
Select the correct answer.
A temporal table must have two period columns, one for the start time, one for the end time.
If you include the HIDDEN keyword when specifying the period columns on a temporal table, these columns cannot be included in a SELECT query.
The FOR SYSTEM_TIME clause is used to query historical data.
You can add system-versioning to a memory-optimized table.
The history table for a system-versioned memory-optimized table is stored on disk.

Lab: Using Advanced Table Designs

Scenario

You are a database developer for Adventure Works who will be designing solutions using corporate databases stored in SQL Server. You have been provided with a set of business requirements and will implement partitioning to archive older data, and data compression to obtain optimal performance and storage from your tables.

Objectives

After completing the lab exercises, you will be able to:

- Create partitioned tables.
- Compress data in tables.

Estimated Time: 30 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Partitioning Data

Scenario

You have created the tables for your business analyst, but believe that the solution could be improved by implementing additional functionality. You will implement partitioned tables to move historical content to an alternative partition.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Create the HumanResources Database
3. Implement a Partitioning Strategy
4. Test the Partitioning Strategy

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab03\Starter** folder as Administrator.

► Task 2: Create the HumanResources Database

1. Using SSMS, connect to **MIA-SQL** using Windows authentication.
2. 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.
3. Create the **HumanResources** database.

► Task 3: Implement a Partitioning Strategy

1. In Solution Explorer, open the query **Lab Exercise 2.sql**.
2. Create four filegroups for the **HumanResources** database: **FG0, FG1, FG2, FG3**.

3. Create a partition function named **pfHumanResourcesDates** in the **HumanResources** database to partition the data by dates.
4. Using the partition function, create a partition scheme named **psHumanResources** to use the four filegroups.
5. Create a **Timesheet** table that will use the new partition scheme.
6. Insert some data into the **Timesheet** table.

► Task 4: Test the Partitioning Strategy

1. In Solution Explorer, open the query **Lab Exercise 3.sql**.
2. Type and execute a Transact-SQL SELECT statement that returns all of the rows from the **Timesheet** table, along with the partition number for each row. You can use the **\$PARTITION** function to achieve this.
3. Type and execute a Transact-SQL statement to view partition metadata.
4. Create a staging table called **Timesheet_Staging** on **FG1**. This should be identical to the **Timesheet** table.
5. Add a check constraint to the **Timesheet_Staging** table to ensure that the values in the **RegisteredStartTime** column meet the following criteria:
 - All values must be greater than or equal to 2011-10-01 00:00.
 - All values must be less than 2012-01-01 00:00.
 - No values can be NULL.
6. Type a Transact-SQL statement to switch out the data in the partition on the filegroup **FG1** to the table **Timesheet_Staging**. Use the **\$PARTITION** function to retrieve the partition number.
7. View the metadata for the partitioned table again to see the changes, and then write and execute a SELECT statement to view the rows in the **Timesheet_Staging** table.
8. Type a Transact-SQL statement to merge the first two partitions, using the value **2011-10-01 00:00**.
9. View the metadata for the partitioned table again to see the changes.
10. Type a Transact-SQL statement to make **FG1** the next used filegroup for the partition scheme.
11. Type a Transact-SQL statement to split the first empty partition, using the value **2012-07-01 00:00**.
12. Type and execute a Transact-SQL statement to add two rows for the new period.
13. View the metadata for the partitioned table again to see the changes.

Results: At the end of this lab, the timesheet data will be partitioned to archive old data.

Exercise 2: Compressing Data

Scenario

The business analyst is satisfied with the partitioning concept you have applied to the Timesheet table. To put the table into production, you will rework the partition to widen the time boundaries to accommodate data over a number of years. After you have populated the Timesheet table, you will decide which compression type to apply to each partition.

The main tasks for this exercise are as follows:

1. Create Timesheet Table for Compression
2. Analyze Storage Savings with Compression
3. Compress Partitions

► Task 1: Create Timesheet Table for Compression

1. In Solution Explorer, open the query **Lab Exercise 4.sql**.
2. Type and execute a T-SQL SELECT statement that drops the **Payment.Timesheet** table.
3. Type and execute a T-SQL SELECT statement that drops the **psHumanResources** partition scheme.
4. Type and execute a T-SQL SELECT statement that drops the **pfHumanResourcesDates** partition function.
5. Type and execute a T-SQL SELECT statement that creates the **pfHumanResourcesDates** partition function, using **RANGE RIGHT** for the values: **2012-12-31 00:00:00.000**, **2014-12-31 00:00:00.000**, and **2016-12-31 00:00:00.000**.
6. Type and execute a T-SQL SELECT statement that creates the **pfHumanResourcesDates** partition scheme, using the filegroups **FG0**, **FG2**, **FG3**, and **FG1**.
7. Type and execute a T-SQL SELECT statement that creates a **Payment.Timesheet** table, using the **pfHumanResourcesDates** partition scheme.
8. Type and execute a T-SQL SELECT statement that adds staff to three shifts, over the course of six years. Exclude weekend dates.

► Task 2: Analyze Storage Savings with Compression

1. In Solution Explorer, open the query **Lab Exercise 5.sql**.
2. Type and execute a T-SQL SELECT statement to view the partition metadata for the **Payment.Timesheet** table.
3. Type and execute a T-SQL SELECT statement to view the estimated savings when applying **ROW** and **PAGE** compression on the **Payment.Timesheet** table.

► Task 3: Compress Partitions

1. In Solution Explorer, open the query **Lab Exercise 6.sql**.
2. Type and execute a T-SQL SELECT statement to partition the **Payment.Timesheet** table. Partitions **FG0** and **FG1** should use **ROW** compression, and partitions **FG2** and **FG3** should use **PAGE** compression.
3. Close SSMS without saving anything.

Results: At the end of this lab, the Timesheet table will be populated with six years of data, and will be partitioned and compressed.

Question: Discuss scenarios that you have experienced where you think partitioning would have been beneficial. Have you worked with databases that could have had older data archived? Were the databases large enough to split the partitions across physical drives for better performance, or to quicken the backup process? Furthermore, could any of this data be compressed? Give reasons for your answers.

Module Review and Takeaways

In this module, you learned:

- How to plan and implement partitioning.
- How to apply data compression to reduce storage of your data, and increase query performance.
- The benefits of using temporal tables to record all changes to your data.

 **Best Practice:** One of the disadvantages of partitioning is that it can be complicated to set up. However, you can use the Developer Edition to replicate your production systems and test your partitioning scenario before applying it in your live environment. As with any major database changes, it is always recommended that you take a backup before applying these changes.

Review Question(s)

Question: What are the advantages of using system-versioning versus a custom-built application to store data changes?

Module 4

Ensuring Data Integrity Through Constraints

Contents:

Module Overview	4-1
Lesson 1: Enforcing Data Integrity	4-2
Lesson 2: Implementing Data Domain Integrity	4-6
Lesson 3: Implementing Entity and Referential Integrity	4-11
Lab: Ensuring Data Integrity Through Constraints	4-22
Module Review and Takeaways	4-25

Module Overview

The quality of data in your database largely determines the usefulness and effectiveness of applications that rely on it—the success or failure of an organization or a business venture could depend on it. Ensuring data integrity is a critical step in maintaining high-quality data.

You should enforce data integrity at all levels of an application from first entry or collection through storage. Microsoft® SQL Server® data management software provides a range of features to simplify the job.

Objectives

After completing this module, you will be able to:

- Describe the options for enforcing data integrity, and the levels at which they should be applied.
- Implement domain integrity through options such as check, unique, and default constraints.
- Implement referential integrity through primary and foreign key constraints.

Lesson 1

Enforcing Data Integrity

Data integrity refers to the consistency and accuracy of data that is stored in a database. An important step in database planning is deciding the best way to enforce this.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how data integrity checks apply across different layers of an application.
- Describe the difference between domain and referential integrity.
- Explain the available options for enforcing each type of data integrity.

Data Integrity Across Application Layers

Application Levels

Applications often have a three-tier hierarchical structure. This keeps related functionality together and improves the maintainability of code, in addition to improving the chance of code being reusable. Common examples of application levels are:

- User interface level.
- Middle tier (sometimes referred to as business logic).
- Data tier.

You can implement data integrity checks at each of these levels.

User Interface Level

There are some advantages of enforcing integrity at the user interface level. The responsiveness to the user may be higher because it can trap minor errors before any service requests from other layers of code. Error messages may be clearer because the code is more aware of which user action caused the error.

The main disadvantage of enforcing integrity at the user interface level is that more than a single application might have to work with the same underlying data, and each application might enforce the rules differently. It is also likely to require more lines of code to enforce business rule changes than may be required at the data tier.

Middle Tier

Many integrity issues highlighted by the code are implemented for the purposes of business logic and functional requirements, as opposed to checking the nonfunctional aspects of the requirements, such as whether the data is in the correct format. The middle tier is often where the bulk of those requirements exist in code, because they can apply to more than one application. In addition, multiple user interfaces often reuse the middle tier. Implementing integrity at this level helps to avoid different user interfaces applying different rules and checks at the user interface level. At this level, the logic is still quite aware of

- Applications are often layered in a hierarchy
- Integrity can be enforced at each level:
 - User interface tier
 - Middle tier
 - Data tier
- It is likely to be found at all tiers:
 - Leading to the complexity of keeping all the constraint management functional and nonfunctional code synchronised

the functions that cause errors, so the error messages generated and returned to the user can still be quite specific.

It is also possible for integrity checks enforced only in the middle tier to compromise the integrity of the data through a mixture of transactional inconsistencies due to optimistic locking, and race conditions, caused by the multithreaded nature of programming models these days. For example, it might seem easy to check that a customer exists and then place an order for that customer. Consider, though, the possibility that another user could remove the customer between the time that you check for the customer's existence and the time that you record the order. The requirement for transactional consistency leads to the necessity for relational integrity of the data elements, which is where the services of a data layer become imperative.

Data Tier

The advantage of implementing integrity at the data tier is that upper layers cannot bypass it. In particular, multiple applications accessing the data simultaneously cannot compromise its quality—there may even be multiple users connecting through tools such as SQL Server Management Studio (SSMS). If referential integrity is not enforced at the data tier level, all applications and users need to individually apply all the rules and checks themselves to ensure that the data is correct.

One of the issues with implementing data integrity constraints at the data tier is the separation between the user actions that caused the errors to occur, and the data tier. This can lead to error messages being precise in describing an issue, but difficult for an end user to understand unless the programmer has ensured that appropriate functional metadata is passed between the system tiers. The cryptic nature of the messages produced by the data tier has to be reprocessed by upper layers of code before presentation to the end user.

Multiple Tiers

The correct solution in most situations involves applying rules and checks at multiple levels. However, the challenge with this approach is in maintaining consistency between the rules and checks at different application levels.

Types of Data Integrity

There are three basic forms of data integrity commonly enforced in database applications: **domain** integrity, **entity** integrity, and **referential** integrity.

Domain Integrity

At the lowest level, SQL Server applies constraints for a domain (or column) by limiting the choice of data that can be entered, and whether nulls are allowed. For example, if you only want whole numbers to be entered, and don't want alphabetic characters, specify the INT (integer) data type.

Equally, assigning a TINYINT data type ensures that only values from 0 to 255 can be stored in that column.

A check constraint can specify acceptable sets of data values, and what default values will be supplied in the case of missing input.

- Domain Integrity
 - Defines the allowed values in columns
- Entity Integrity
 - Primary key uniquely identifies each row within a table
- Referential integrity
 - Defines the relationship between tables

Entity Integrity

Entity or table integrity ensures that each row within a table can be identified uniquely. This column (or columns in the case of a composite key) is known as the table's primary key. Whether the primary key value can be changed or whether the whole row can be deleted depends on the level of integrity that is required between the primary key and any other tables, based on referential integrity.

This is where the next level of integrity comes in, to ensure the changes are valid for a given relationship.

Referential Integrity

Referential integrity ensures that the relationships among the primary keys (in the referenced table) and foreign keys (in the referencing tables) are maintained. You are not permitted to insert a value in the referencing column that does not exist in the referenced column in the target table. A row in a referenced table cannot be deleted, nor can the primary key be changed, if a foreign key refers to the row unless a form of cascading action is permitted. You can define referential integrity relationships within the same table or between separate tables.

As an example of referential integrity, you may have to ensure that an order cannot be placed for a nonexistent customer.

Options for Enforcing Data Integrity

The table on the slide summarizes the mechanisms that SQL Server provides for enforcing data integrity.

Data Types

The first option for enforcing data integrity is to ensure that only the correct type of data is stored in a given column. For example, you cannot place alphabetic characters into a column that has been defined as storing integers.

The choice of a data type will also define the permitted range of values that can be stored. For example, the **smallint** data type can only contain values from -32,768 to 32,767.

For XML data (which is discussed in Module 14) XML schemas can be used to further constrain the data that is held in the XML data type.

- Data type: defines the type of data that can be stored in a column
- Nullability: determines whether a value must be present in a column
- Constraints: defines rules that limit the values that can be stored in a column, or how values in different columns must be related; also default values
- Triggers: define code that is executed automatically when data in a table is modified

Null-ability Constraint

This determines whether a column can store a null value, or whether a value must be provided. This is often referred to as whether a column is mandatory or not.

Default Values

If a column has been defined to not allow nulls, then a value must be provided whenever a new row is inserted. With a default value, you can ignore the column during input and a specific value will be inserted into the column when no value is supplied.

Check Constraint

Constraints are used to limit the permitted values in a column further than the limits that the data type, null-ability and a default provides. For example, a **tinyint** column can have values from 0 to 255. You might decide to further constrain the column so that only values between 1 and 9 are permitted.

You can also apply constraints at the table level and enforce relationships between the columns of a table. For example, you might have a column that holds an order number, but it is not mandatory. You might then add a constraint that specifies that the column must have a value if the Salesperson column also has a value.

Triggers

Triggers are procedures, somewhat like stored procedures, that are executed whenever specific events such as an INSERT or UPDATE occur. Triggers are executed automatically whenever a specific event occurs. Within the trigger code, you can enforce more complex rules for integrity. Triggers are discussed in Module 11.

Objects from Earlier Versions

Early versions of SQL Server supported objects called rules and defaults. Note that defaults were a type of object and not the same as DEFAULT constraints. Defaults were separate objects that were then bound to columns. They were reused across multiple columns.

These objects have been deprecated because they were not compliant with Structured Query Language (SQL) standards. Code that is based on these objects should be replaced. In general, you should replace rules with CHECK constraints and defaults with DEFAULT constraints.

Sequencing Activity

Put the following constraint types in order by numbering each to indicate the order of importance to minimize constraint checking effort.

Steps
Specify data type.
Indicate column null-ability.
Indicate column default value.
Indicate a check constraint.
Write a trigger to control the column contents.

Lesson 2

Implementing Data Domain Integrity

Domain integrity limits the range and type of values that can be stored in a column. It is usually the most important form of data integrity when first designing a database. If domain integrity is not enforced, processing errors can occur when unexpected or out-of-range values are encountered.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how you can use data types to enforce basic domain integrity.
- Describe the default value's null-ability when entering the domain set as a valid value.
- Describe how you can use an additional DEFAULT constraint to provide a non-null default value for the column.
- Describe how you can use CHECK constraints to enforce domain integrity beyond a null and default value.

Data Types

Choosing an appropriate data type for each column is one of the most important decisions that you must make when you are designing a table as part of a database. Data types were discussed in detail in Module 2.

You can assign data types to a column by using one of the following methods:

- Using SQL Server system data types.
- Creating alias data types that are based on system data types.
- Creating user-defined data types based on data types created within the Microsoft .NET Framework common language runtime (CLR).

- Choosing data types is an important decision when designing tables
- They can be assigned by using:
 - System data types
 - Alias data types
 - User-defined data types

System Data Types

SQL Server supplies a system-wide range of built-in data types. Choosing a data type determines both the type of data that can be stored and the range of values that is permitted within the column.

Alias Data Types

Inconsistency between column data types can cause problems. The situation is exacerbated when more than one person has designed the tables. For example, you may have several tables that store the weight of a product that was sold. One column might be defined as decimal(18,3), another column might be defined as decimal(12,2), and a third column might be defined as decimal(16,5). For consistency, alias data types can create a data type called ProductWeight, and define it as decimal(18,3), which you can then implement as the data type for all of the columns. This can lead to more consistent database designs.

An additional advantage of alias data types is that code generation tools can create more consistent code when the tools have the additional information about the data types that alias data types provide. For example, you could decide to have a user interface design program that always displayed and/or prompted for product weights in a specific way.

User-Defined Data Types

The addition of managed code to SQL Server 2005 made it possible to create entirely new data types. Although alias data types are user-defined, they are still effectively subsets of the existing system data types. With user-defined data types that are created in managed code, you can define not only the data that is stored in a data type, but also the behavior of the data type. For example, you can design a JPEG data type. Besides designing how it will store images, you can define it to be updated by calling a predesigned method. Designing user-defined data types is discussed in more detail in Module 13.

DEFAULT Constraints

A DEFAULT constraint provides a value for a column when no value is specified in the statement that inserted the row. You can view the existing definition of DEFAULT constraints by querying the sys.default_constraints view.

DEFAULT Constraint

Sometimes a column is mandatory—that is, a value must be provided. However, the application or program that is inserting the row might not provide a value. In this case, you may want to apply a value to ensure that the row will be inserted.

DEFAULT constraints are associated with a table column. They are used to provide a default value for the column when the user does not supply a value. The value is retrieved from the evaluation of an expression and the data type that the expression returns must be compatible with the data type of the column.

- Default constraints
 - Provide default values for columns
 - Used if INSERT provides no column value
 - Must produce data compatible with the data type for the column

Nullable Columns and DEFAULT Constraint Coexistence

Without a DEFAULT constraint, if no value is provided for the column in the statement that inserted the row, the column value would be set to NULL. If a NOT NULL constraint has been applied to the column, the insertion would fail. With a DEFAULT constraint, the default value would be used instead. DEFAULT constraints might be used to insert the current date or the identity of the user inserting the data into the table.

 **Note:** If the statement that inserted the row explicitly inserted NULL, the default value would not be used.

Named Constraints

SQL Server does not require you to supply names for constraints that you create. If a name is not supplied, SQL Server will automatically generate a name. However, the names that are generated are not very intuitive. Therefore, it is generally considered a good idea to provide names for constraints as you create them—and to do so using a naming standard.

A good example of why naming constraints is important is that, if a column needs to be deleted, you must first remove any constraints that are associated with the column. Dropping a constraint requires you to provide a name for the constraint that you are dropping. Having a consistent naming standard for constraints helps you to know what that name is likely to be, rather than having to execute a query to find the name. Locating the name of a constraint would involve querying the sys.constraints system view, searching in Object Explorer, or selecting the relevant data from the INFORMATION_SCHEMA.CONSTRAINTS catalogue view.

The following code shows a named default constraint:

Named Default Constraint

```
CREATE TABLE dbo.SalesOrder
(
    OpportunityID int,
    ReceivedDate date NOT NULL
        CONSTRAINT DF_SalesOrder_Date DEFAULT (SYSDATETIME()),
    ProductID int NOT NULL,
    SalespersonID int NOT NULL
);
```

CHECK Constraints

A CHECK constraint limits the values that a column can accept by controlling the values that can be put in the column.

After determining the data type, and whether it can be null, you may want to further restrict the values that can be placed into the column. For example, you might decide that a **varchar(7)** column must be five characters long if the first character is the letter A.

More commonly, CHECK constraints are used as a form of "sanity" check. For example, you might decide that a salary needs to be within a certain range, or a person's age must be in the range 0 to 130.

Logical Expression

CHECK constraints work with any logical (Boolean) expression that can return TRUE, FALSE, or UNKNOWN. Particular care must be given to any expression that could have a NULL return value. CHECK constraints reject values that evaluate to FALSE, but not an unknown return value, which is what NULL evaluates to.

- Check constraints
 - Limit the values that are accepted in a column
 - Only rejects FALSE outcomes
 - NULL evaluates to UNKNOWN and not FALSE
 - Can be defined at table level to refer to multiple columns

Create table with a check constraint.

Check Constraint

```
CREATE TABLE Sales.opportunity
(
    opportunityID int NOT NULL,
    requirements nvarchar(50) NOT NULL,
    salespersonID int NOT NULL,
    rating int NOT NULL
        CONSTRAINT CK_Opportunity_Rating1to4
        CHECK (Rating BETWEEN 1 AND 4)
);
```

For more information about column level constraints, see *column_constraint (Transact SQL)* in Microsoft Docs:

 **column_constraint (Transact SQL)**

<http://aka.ms/b9ty9o>

Table-Level CHECK Constraints

Apart from checking the value in a particular column, you can apply CHECK constraints at the table level to check the relationship between the values in more than a single column from the same table. For example, you could decide that the FromDate column should not have a larger value than theToDate column in the same row.

For more information about table-level constraints, see Microsoft Docs:

 **ALTER TABLE table_constraint**

<http://aka.ms/peaqqm>

Demonstration: Data and Domain Integrity

In this demonstration, you will see how to:

- Enforce data and domain integrity.

Demonstration Steps

Enforce Data and Domain Integrity

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **D:\Demofiles\Mod04\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. On the taskbar, click **Microsoft SQL Server Management Studio**.
5. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL** and then click **Connect**.
6. On the **File** menu, point to **Open**, click **Project/Solution**.
7. In the **Open Project** dialog box, navigate to **D:\Demofiles\Mod04**, click **Demo04.ssmssln**, and then click **Open**.
8. In Solution Explorer, expand the **Queries** folder, and double-click **21 - Demonstration 2A.sql**.

9. Familiarize yourself with the requirement using the code below **Step 1: Review the requirements for a table design.**
10. Place the pseudo code for your findings for the requirements below **Step 2: Determine the data types, null-ability, default and check constraints that should be put in place.**
11. Highlight the code below **Step 3: Check the outcome with this proposed solution**, and click **Execute**.
12. Highlight the code below **Step 4: Execute statements to test the actions of the integrity constraints**, and click **Execute**.
13. Highlight the code below **Step 5: INSERT rows that test the nullability and constraints**, and click **Execute**. Note the errors.
14. Highlight the code below **Step 6: Query sys.sysconstraints to see the list of constraints**, and click **Execute**.
15. Highlight the code below **Step 7: Explore system catalog views through the INFORMATION_SCHEMA owner**, and click **Execute**.
16. Close SQL Server Management Studio, without saving any changes.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? When you have a check constraint on a column, it is not worth having a NOT NULL constraint because any nulls will be filtered out by the check constraint.	

Lesson 3

Implementing Entity and Referential Integrity

It is important to be able identify rows within tables uniquely and to be able to establish relationships across tables. For example, if you have to ensure that an individual can be identified as an existing customer before an order can be placed, you can enforce this by using a combination of entity and referential integrity.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how PRIMARY KEY constraints enforce entity integrity.
- Describe the use of the IDENTITY property for Primary Keys.
- Describe the use of sequences for primary keys.
- Describe how UNIQUE constraints are sometimes used instead of PRIMARY KEY constraints.
- Explain how FOREIGN KEY constraints enforce referential integrity.
- Describe how data changes cascade through relationships.
- Explain the common considerations for constraint checking.

PRIMARY KEY Constraints

PRIMARY KEY constraints were introduced in Module 2, and are used to uniquely identify each row in a table. They must be unique, not NULL, and may involve multiple columns. SQL Server will internally create an index to support the PRIMARY KEY constraint.

Remember that the term "candidate key" is used to describe the column or combination of columns that could uniquely identify a row of data within a table. None of the columns that are part of a candidate key are permitted to be nullable.

In the following example, the OpportunityID column has been chosen as the primary key.

As with other types of constraints, even though a name is not required when defining a PRIMARY KEY constraint, it is preferable to choose a name for the constraint, rather than leaving SQL Server to do so.

- Primary keys
 - Are used to uniquely identify a row in a table
 - Must be unique and not NULL
 - May involve multiple columns that form a composite key

The following code shows a primary key constraint:

Primary Key Constraint

```
CREATE TABLE sales.opportunity
(OpportunityID int NOT NULL
 CONSTRAINT PK_Opportunity PRIMARY KEY,
 Requirements nvarchar(50) NOT NULL,
 ReceivedDate date NOT NULL,
 SalespersonID int NULL,
 Rating int NOT NULL
);
```

For more information about how to create primary keys, see *Create Primary Keys* in the SQL Server Technical Documentation:

Create Primary Keys

<http://aka.ms/ulm1g6>

UNIQUE Constraints

A UNIQUE constraint indicates that the column or combination of columns is unique. One row can be NULL (if the column null-ability permits this). SQL Server will internally create an index to support the UNIQUE constraint.

For example, in Spain, all Spanish citizens over the age of 14 receive a national identity document called a Documento Nacional de Identidad (DNI). It is a unique number in the format 99999999-X where 9 is a digit and X is a letter used as a checksum of the digits. People from other countries who need a Spanish identification number are given a Número de Identidad de Extranjero (NIE), which has a slightly different format of X-99999999-X.

- Unique constraints
 - Ensure that values in a column are unique
 - One row may have a NULL value
 - You can have multiple unique columns

If you were storing a tax identifier for employees in Spain, you would store one of these values, include a CHECK constraint to make sure that the value was in one of the two valid formats, and have a UNIQUE constraint on the column that stores these values. Note that this may be unrelated to the fact that the table has another unique identifier, such as EmployeeID, used as a primary key for the table.

As with other types of constraints, even though a name is not required when defining a UNIQUE constraint, you should choose a name for the constraint rather than leaving SQL Server to do so.

The following code shows a unique constraint:

Unique Constraint

```
CREATE TABLE Sales.Opportunity
(
    OpportunityID int NOT NULL
        CONSTRAINT PK_Opportunity PRIMARY KEY,
    Requirements nvarchar(50) NOT NULL
        CONSTRAINT UQ_Opportunity_Requirements UNIQUE,
    ReceivedDate date NOT NULL
);
```

NULL and UNIQUE

Although it is possible for a column that is required to be unique to be null, a null key value is only valid for a single row. In practice, this means that nullable unique columns are rare. A UNIQUE constraint is used to ensure that more than one row does not have a single value, including NULL.

Create Unique Constraints

<http://aka.ms/npf0a6>

IDENTITY Constraints

It is common to need automatically generated numbers for an **integer** primary key column. The IDENTITY property on a database column indicates that an INSERT statement will not provide the value for the column; instead, SQL Server will provide it automatically.

IDENTITY is a property that is typically associated with **int** or **bigint** columns that provide automated generation of values during insert operations. You may be familiar with auto-numbering systems or sequences in other database engines. IDENTITY columns are not identical to these, but you can use them to replace the functionality from those other database engines.

- IDENTITY property
 - Automatically generates column values
 - You can specify a seed (starting number) and an increment
 - Default seed and increment are both 1
 - SCOPE IDENTITY(), @@IDENTITY return current value

Adding the IDENTITY Qualifier to a Table Column

When you specify the identity property **CustomerID INT IDENTITY (1, 10)** as a qualifier for a numeric column, you specify a seed (1, in this example) and an increment (10, in this example). The seed is the starting value. The increment is how much the value goes up each time it is incremented. Both seed and increment default to a value of 1 if they are not specified.

The following code adds the IDENTITY property to the OpportunityID column:

IDENTITY property

```
CREATE TABLE Sales.Opportunity
(
    OpportunityID int NOT NULL IDENTITY(1,1),
    Requirements nvarchar(50) NOT NULL,
    ReceivedDate date NOT NULL,
    SalespersonID int NULL
);
```

Adding Rows with Explicit Values for the Identity Column

Although explicit inserts are not normally permitted for columns that have an IDENTITY property, you can explicitly insert values. You can do this by using a table setting option, **SET IDENTITY_INSERT customer ON**. With this option, you can explicitly insert values into the column with the IDENTITY property within the customer table. Remember to switch the automatic generation back on after you have inserted exceptional rows.



Note: Having the IDENTITY property on a column does not ensure that the column is unique. Define a UNIQUE constraint to guarantee that values in the column will be unique.

Retrieving the Inserted Identity Value

After inserting a row into a table, you often have to know the value that was placed into the column with the IDENTITY property. The syntax **SELECT @@IDENTITY** returns the last identity value that was used within the session, in any scope. This can be useful when triggers perform inserts on another table with an IDENTITY column as part of an INSERT statement.

For example, if you insert a row into a customer table, the customer might be assigned a new identity value. However, if a trigger on the customer table caused an entry to be written into an audit logging table, when inserts are performed, the @@IDENTITY variable would return the identity value from the audit logging table, rather than the one from the customer table.

To deal with this effectively, the **SCOPE_IDENTITY()** function was introduced. It provides the last identity value but only within the current scope. In the previous example, it would return the identity value from the customer table.

Retrieving the Identities of a Multirow INSERT

Another complexity relates to multirow inserts. In this situation, you may want to retrieve the IDENTITY column value for more than one row at a time. Typically, this would be implemented by the use of the OUTPUT clause on the INSERT statement.



IDENTITY (Property) (Transact-SQL)

<http://aka.ms/ik8k5i>

Working with Sequences

Sequences

Sequences are another way of creating values for insertion into a column as sequential numbers.

However, unlike IDENTITY properties, sequences are not tied to any specific table. This means that you could use a single sequence to provide key values for a group of tables.

Sequences can be cyclic. They can return to a low value when a specified maximum value has been exceeded.

In the example, a sequence called BookingID is created in the Booking schema. This means that you can have the same name for different sequences within different schemas. The sequence is defined as generating integer values. By default, sequences generate bigint values.

Values from sequences are retrieved by using the NEXT VALUE FOR clause. In the example, the sequence is being used to provide the default value for the FlightBookingID column in the Booking.FlightBooking table.

Sequences are created by the CREATE SEQUENCE statement, modified by the ALTER SEQUENCE statement, and deleted by the DROP SEQUENCE statement.

- Sequence objects:
 - Are user-defined, schema-bound objects
 - Are not tied to any particular table
 - Can be used to ease migration from other database engines

Other database engines provide sequence values, so the addition of sequence support in SQL Server 2012 and SQL Server 2014 can assist with migrating code to SQL Server from other database engines.

A range of sequence values can be retrieved in a single call via the sp_sequence_get_range system stored procedure. There are also options to cache sets of sequence values to improve performance. However, when a server failure occurs, the entire cached set of values is lost.

Values that are retrieved from the sequence are not available for reuse. This means that gaps can occur in the set of sequence values.

The following code shows how to create and use a sequence object:

SEQUENCE

```
CREATE SEQUENCE Booking.Booking1D AS INT
    START WITH 20001
    INCREMENT BY 10;
GO

CREATE TABLE Booking.FlightBooking
(
    FlightBooking1D INT NOT NULL PRIMARY KEY CLUSTERED DEFAULT
        (NEXT VALUE FOR Booking.BookingID)
    ...
)
```

For more information about sequences, see *Sequence Properties (General Page)* in Microsoft Docs:

Sequence Properties (General Page)

<http://aka.ms/dga6do>

Demonstration: Sequences Demonstration

In this demonstration, you will see how to:

- Work with identity constraints, create a sequence, and use a sequence to provide key values for two tables.

Demonstration Steps

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **D:\Demofiles\Mod04\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. On the taskbar, click **Microsoft SQL Server Management Studio**.
5. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL** and then click **Connect**.
6. On the **File** menu, point to **Open**, click **Project/Solution**.
7. In the **Open Project** dialog box, navigate to **D:\Demofiles\Mod04**, click **Demo04.ssmssln**, and then click **Open**.
8. In Solution Explorer, double-click the **31 - Demonstration 3A.sql** script file.

9. Highlight the code below **Step 1: Open a new query window to the tempdb database**, and click **Execute**.
10. Highlight the code below **Step 2: Create the dbo.Opportunity table**, and click **Execute**.
11. Highlight the code below **Step 3: Populate the table with two rows**, and click **Execute**.
12. Highlight the code below **Step 4: Check the identity values added**, and click **Execute**.
13. Highlight the code below **Step 5: Try to insert a specific value for OpportunityID**, and click **Execute**. Note the error.
14. Highlight the code below **Step 6: Add a row without a value for LikelyClosingDate**, and click **Execute**.
15. Highlight the code below **Step 7: Query the table to see the value in the LikelyClosingDate column**, and click **Execute**.
16. Highlight the code below **Step 8: Create 3 Tables with separate identity columns**, and click **Execute**.
17. Highlight the code below **Step 9: Insert some rows into each table**, and click **Execute**.
18. Highlight the code below **Step 10: Query the 3 tables in a single view and note the overlapping ID values**, then click **Execute**.
19. Highlight the code below **Step 11: Drop the tables**, and click **Execute**.
20. Highlight the code below **Step 12: Create a sequence to use with all 3 tables**, and click **Execute**.
21. Highlight the code below **Step 13: Recreate the tables using the sequence for default values**, and click **execute**.
22. Highlight the code below **Step 14: Reinsert the same data**, and click **Execute**.
23. Highlight the code below **Step 15: Note the values now appearing in the view**, and click **execute**.
24. Highlight the code below **Step 16: Note that sequence values can be created on the fly**, and click **Execute**.
25. Highlight the code below **Step 17: Re-execute the same code and note that the sequence values**, and click **Execute**.
26. Highlight the code below **Step 18: Note that when the same entry is used multiple times in a SELECT statement, that the same value is used**, and click **Execute**.
27. Highlight the code below **Step 19: Fetch a range of sequence values**, and click **Execute**.
28. Close SQL Server Management Studio, without saving any changes.

FOREIGN KEY Constraints

The FOREIGN KEY constraint was introduced in Module 2. Foreign keys are used to link two tables, and create a relationship between them. As an example, a relationship could be to ensure that a customer exists for any products that are placed in the orders table. This is automatically created when you specify a foreign key in the orders table referring to the primary key in the customers table.

- Foreign key constraints:
 - Are used to enforce relationships between tables
 - Check the existence of parent when inserting child
 - Check the existence of children when deleting parent
 - Can be self-referencing

The column that the foreign key references must be defined either as a primary key in the linked table, or the column must have a unique constraint. You cannot create a link to a NULL value, so the unique constraint must have a NOT NULL constraint.

In Module 2, you also saw that the target table can be the same table, known as a self-referencing relationship.

As with other types of constraints, even though a name is not required when defining a FOREIGN KEY constraint, you should provide a name rather than leaving SQL Server to do so.

Defining a foreign key constraint.

Foreign Key Constraint

```
CREATE TABLE sales.Opportunity
(
    OpportunityID int NOT NULL CONSTRAINT PK_Opportunity PRIMARY KEY,
    Requirements nvarchar(50) NOT NULL,
    SalespersonID int NULL
        CONSTRAINT FK_Opportunity_Salesperson
        FOREIGN KEY REFERENCES sales.Salesperson(SalespersonID)
);
```

WITH NOCHECK Option

When you add a FOREIGN KEY constraint to a column (or columns) in a table, SQL Server will check the data that is already in the column to make sure that the reference to the target table is valid. However, if you specify WITH NOCHECK, SQL Server does not apply the check to existing rows and will only check the reference in future when rows are inserted or updated. The WITH NOCHECK option can be applied to other types of constraints, too.

Permission Requirement

Before you can place a FOREIGN KEY constraint on a table, you must have CREATE TABLE and ALTER TABLE permissions.

The REFERENCES permission on the target table avoids the situation where another user could place a reference to one of your tables, leaving you unable to drop or substantially change your own table until the other user removed that reference. However, in terms of security, remember that providing REFERENCES permission to a user on a table for which they do not have SELECT permission does not totally prevent them from working out what the data in the table is. This might be done by a brute force attempt that involves trying all possible values.

 **Note:** Changes to the structure of the referenced column are limited while it is referenced in a FOREIGN KEY. For example, you cannot change the size of the column when the relationship is in place.

 **Note:** The NOCHECK applies to a foreign key constraint in addition to other constraints defined on the table. This prevents the constraint from checking data that is already present. This is useful if a constraint should be applied to all new records, but existing data does not have to meet the criteria.

For more information about defining foreign keys, see *Create Foreign Key Relationships* in Microsoft Docs:

 **Create Foreign Key Relationships**

<http://aka.ms/l2m22c>

Cascading Referential Integrity

The FOREIGN KEY constraint includes a facility to make any change to a column value that defines a UNIQUE or PRIMARY KEY constraint to propagate the change to any foreign key values that reference it. This action is referred to as cascading referential integrity.

By using cascading referential integrity, you can define the actions that SQL Server takes when a user tries to update or delete a key column (or columns) to which a FOREIGN KEY constraint makes reference.

The action to be taken is separately defined for UPDATE and DELETE actions and can have four values:

1. **NO ACTION** is the default. For example, if you attempt to delete a customer and there are orders for the customer, the deletion will fail.
2. **CASCADE** makes the required changes to the referencing tables. If the customer is being deleted, his or her orders will also be deleted. If the customer primary key is being updated (although note that this is undesirable), the customer key in the orders table will also be updated so that the orders still refer to the correct customer.
3. **SET DEFAULT** causes the values in the columns in the referencing table to be set to their default values. This provides more control than the SET NULL option, which always sets the values to NULL.
4. **SET NULL** causes the values in the columns in the referencing table to be nullified. For the customer and orders example, this means that the orders would still exist, but they would not refer to any customer.

 **Cascading referential integrity**

- Is controlled by the CASCADE option of the FOREIGN KEY constraint
 - NO ACTION (default): return error and rollback operation
 - CASCADE: update foreign keys in referencing tables. Delete rows in referencing tables
 - SET DEFAULT: set foreign keys in referencing tables to default values
 - SET NULL: set foreign keys in referencing tables to NULL

Caution

Although cascading referential integrity is easy to set up, you should be careful when using it within database designs.

For example, if you used the CASCADE option in the example above, would it really be okay for the orders for the customer to be removed when you remove the customer? When you remove the customer, you also delete their orders. There might be other tables that reference the orders table (such as order details or invoices), and these would also be removed if they had a cascade relationship set up.

Primary and Foreign Key Constraints

<http://aka.ms/o5f11a>

Considerations for Constraint Checking

There are a few considerations when you are working with constraints.

Naming

Specify meaningful names for constraints rather than leaving SQL Server to select a name. SQL Server provides complicated system-generated names. Often, you have to refer to constraints by name. Therefore, it is better to have chosen them yourself using a consistent naming convention.

- Give constraints meaningful names
- Constraints can be created, changed, and dropped without having to drop and recreate the table
- Perform error checking in your applications and transactions
- Referential constraints can be also be suspended
 - To improve performance during large batch jobs
 - To avoid checking existing data when you add new constraints to a table containing valid data
 - You must have the name that you or the system supplied to suspend the checks

Changing Constraints

You can create, alter, or drop constraints without having to drop and recreate the underlying table.

You use the ALTER TABLE statement to add, alter, or drop constraints.

Error Checking in Applications

Even though you have specified constraints in your database layer, you may also want to check the same logic in higher layers of code. Doing so will lead to more responsive systems because they will go through fewer layers of code. It will also provide more meaningful errors to users because the code is closer to the business-related logic that led to the errors. The challenge is in keeping the checks between different layers consistent.

High-Performance Data Loading or Updates

When you are performing bulk loading or updates of data, you can often achieve better performance by disabling CHECK and FOREIGN KEY constraints while performing the bulk operations and then re-enabling them afterwards, rather than having them checked row by row during the bulk operation.

Demonstration: Entity and Referential Integrity

In this demonstration, you will see how to:

- Define entity integrity for tables.
- Define referential integrity for tables.
- Define cascading actions to relax the default referential integrity constraint.

Demonstration Steps

Define entity integrity for a table, define referential integrity for a table, and define cascading referential integrity actions for the constraint.

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **D:\Demofiles\Mod04\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. On the taskbar, click **Microsoft SQL Server Management Studio**.
5. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL** and then click **Connect**.
6. On the **File** menu, point to **Open**, click **Project/Solution**.
7. In the **Open Project** dialog box, navigate to **D:\Demofiles\Mod04**, click **Demo04.ssmssln**, and then click **Open**.
8. In Solution Explorer, double-click the **32 - Demonstration 3B.sql** script file.
9. Highlight the code below **Step 1: Open a new query window to tempdb**, and click **Execute**.
10. Highlight the code below **Step 2: Create the Customer and CustomerOrder tables**, and click **Execute**.
11. Highlight the code below **Step 3: Select the list of customers**, and click **Execute**.
12. Highlight the code below **Step 4: Try to insert a CustomerOrder row for an invalid customer**, and click **Execute**. Note the error message.
13. Highlight the code below **Step 5: Try to remove a customer that has an order**, and click **Execute**. Note the error message.
14. Highlight the code below **Step 6: Replace it with a named constraint with cascade**, and click **Execute**.
15. Highlight the code below **Step 7: Select the list of customer orders, try a delete again**, and click **Execute**.
16. Highlight the code below **Step 8: Note how the cascade option caused the orders**, and click **Execute**.
17. Highlight the code below **Step 9: Try to drop the referenced table and note the error**, then click **Execute**. Note the error message.
18. Close SQL Server Management Studio, without saving any changes.

Check Your Knowledge

Question
You want to set up a cascading referential integrity constraint between two tables that has the minimum impact on queries that are only interested in current data rather than historic data. What would you use?
Select the correct answer.
<input type="checkbox"/> ON DELETE CASCADE
<input type="checkbox"/> ON DELETE RESTRICT
<input type="checkbox"/> ON DELETE SET DEFAULT
<input type="checkbox"/> ON DELETE SET NULL

Lab: Ensuring Data Integrity Through Constraints

Scenario

A table named Yield has recently been added to the Marketing schema within the database, but it has no constraints in place. In this lab, you will implement the required constraints to ensure data integrity and, if you have time, test that constraints work as specified.

Column Name	Data Type	Required	Validation Rule
OpportunityID	Int	Yes	Part of the Primary key
ProspectID	Int	Yes	Part of the key—also, prospect must exist
DateRaised	datetime	Yes	Must be today's date
Likelihood	Bit	Yes	
Rating	char(1)	Yes	
EstimatedClosingDate	date	Yes	
EstimatedRevenue	decimal(10,2)	Yes	

Objectives

After completing this lab, you will be able to:

- Use the ALTER TABLE statement to adjust the constraints on existing tables.
- Create and test a DEFAULT constraint.
- Create and test a CHECK constraint.
- Create and test a UNIQUE constraint.
- Create and test a PRIMARY KEY constraint.
- Create and test a Referential Integrity FOREIGN KEY constraint.
- Create and test a CASCADING REFERENTIAL INTEGRITY constraint for a FOREIGN KEY and a PRIMARY KEY.

Estimated Time: 30 Minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Add Constraints

Scenario

You have been given the design for a table called DirectMarketing.Opportunity. You must alter the table with the appropriate constraints, based upon the provided specifications.

The main tasks for this exercise are as follows:

- Review the supporting documentation.
- Alter the DirectMarketing.Opportunity table.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Review Supporting Documentation
3. Alter the Direct Marketing Table

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. On the taskbar, click **File Explorer**.
3. In File Explorer, navigate to the **D:\Labfiles\Lab04\Starter** folder, right-click the **Setup.cmd** file, and then click **Run as administrator**.
4. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

► Task 2: Review Supporting Documentation

- Review the table design requirements that were supplied in the scenario.

► Task 3: Alter the Direct Marketing Table

1. Work through the list of requirements and alter the table to make the columns required, based on the requirements.
2. Work through the list of requirements and alter the table to make columns the primary key, based on the requirements.
3. Work through the list of requirements and alter the table to make columns foreign keys, based on the requirements.
4. Work through the list of requirements and alter the table to add DEFAULT constraints to columns, based on the requirements.

Exercise 2: Test the Constraints

Scenario

You should now test each of the constraints that you designed to ensure that they work as expected.

The main tasks for this exercise are as follows:

- Test the default values and data types.
- Test the primary key.
- Test the foreign key reference on **ProspectID**.

The main tasks for this exercise are as follows:

1. Test the Data Types and Default Constraints
2. Test the Primary Key
3. Test to Ensure the Foreign Key is Working as Expected

► Task 1: Test the Data Types and Default Constraints

- Create a new query for the solution called **ConstraintTesting.sql**. Use this new connection to Adventureworks to insert a row into the opportunity table using the following values, which are organized by the columns as found within the table: [1,1,8,'A','12/12/2013',123000.00].

► Task 2: Test the Primary Key

- Try to add the same row again to confirm that the primary key constraint is working to ensure entity integrity—only unique rows can be added to the table

► Task 3: Test to Ensure the Foreign Key is Working as Expected

- Try to add some data for a prospect that does not exist, to confirm that the foreign key constraint is working, to ensure relational integrity. Only nonunique rows are to be added to the table for foreign key values that are uniquely available in the prospects table.

Results: After completing this exercise, you should have successfully tested your constraints.

Question: Why implement CHECK constraints if an application is already checking the input data?

Question: What are some scenarios in which you might want to temporarily disable constraint checking?

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? A PRIMARY KEY and a UNIQUE constraint are doing the same thing using different code words.	

Module Review and Takeaways



Best Practice: When you create a constraint on a column, if you do not specify a name for the constraint, SQL Server will generate a unique name for it. However, you should always name constraints to adhere to your naming conventions. This makes the constraints easier to identify when they appear in error messages. They are also easier to alter because you don't have to remember any arbitrary numbers that SQL Server uses to make the names unique when it generates constraint names automatically.

Review Question(s)

Question: Would you consider that you need to CHECK constraints if an application is already checking the input data?

Question: What are some scenarios in which you might want to temporarily disable constraint checking?

Module 5

Introduction to Indexes

Contents:

Module Overview	5-1
Lesson 1: Core Indexing Concepts	5-2
Lesson 2: Data Types and Indexes	5-8
Lesson 3: Heaps, Clustered, and Nonclustered Indexes	5-12
Lesson 4: Single Column and Composite Indexes	5-22
Lab: Implementing Indexes	5-26
Module Review and Takeaways	5-29

Module Overview

An index is a collection of pages associated with a table. Indexes are used to improve the performance of queries or enforce uniqueness. Before learning to implement indexes, it is helpful to understand how they work, how effective different data types are when used within indexes, and how indexes can be constructed from multiple columns. This module discusses table structures that do not have indexes, and the different index types available in Microsoft® SQL Server®.

Objectives

After completing this module, you will be able to:

- Explain core indexing concepts.
- Evaluate which index to use for different data types.
- Describe the difference between single and composite column indexes.

Lesson 1

Core Indexing Concepts

Although it is possible for Microsoft SQL Server data management software to read all of the pages in a table when it is calculating the results of a query, doing so is often highly inefficient. Instead, you can use indexes to point to the location of required data and to minimize the need for scanning entire tables. In this lesson, you will learn how indexes are structured and learn the principles associated with the design of indexes. Finally, you will see how indexes can become fragmented over time, and the steps required to resolve this fragmentation.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how SQL Server accesses data.
- Describe the need for indexes.
- Explain the concept of b-tree index structures.
- Explain the concepts of selectivity, density, and index depth.
- Understand why index fragmentation occurs.
- Deduce the level of fragmentation on an index.

How SQL Server Accesses Data

SQL Server can access data in a table by reading all of the pages in the table, which is known as a table scan, or by using index pages to locate the required rows. Each page in an index is 8 kilobytes (KB) in size.

Whenever SQL Server needs to access data in a table, it has to choose between doing a table scan or seeking and reading one or more indexes. SQL Server will choose the option with the least amount of effort to locate the required rows.

You can always resolve queries by reading the underlying table data. Indexes are not required, but accessing data by reading large numbers of pages is considerably slower than methods that use appropriate indexes.

Sometimes SQL Server creates its own temporary indexes to improve query performance. However, doing so is up to the optimizer and beyond the control of the database administrator or programmer; these temporary indexes will not be discussed in this module. Temporary indexes are only used to improve a query plan if no suitable indexing already exists. A table without an index is referred to as a heap table.

In this module, you will consider standard indexes that are created on tables. SQL Server also includes other types of index:

- Integrated full-text search is a special type of index that provides flexible searching of text.
- Spatial indexes are used with the GEOMETRY and GEOGRAPHY data types.

- Table Scan
 - SQL Server reads all table pages
 - Any query can be satisfied by a table scan
 - Will result in the slowest response to a query
 - A table without indexes is called a heap
- Index
 - SQL Server uses index pages to find the desired rows
 - Different types
 - Clustered and nonclustered
 - Rowstore and columnstore

- Primary and secondary XML indexes assist when querying XML data.
- Columnstore indexes are used to speed up aggregate queries against large data sets.

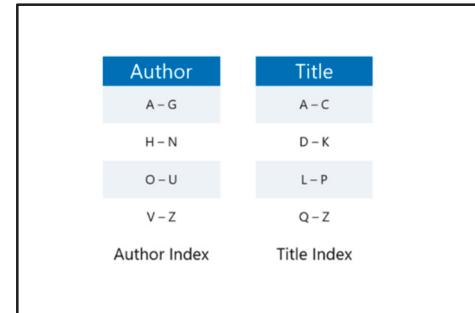
Each of these other index types are discussed in later modules.

The Need for Indexes

Indexes are not described in ANSI Structured Query Language (SQL) definitions. Indexes are considered to be an implementation detail for the vendor. SQL Server uses indexes for improving the performance of queries and for implementing certain constraints.

As mentioned in the last topic, SQL Server can always read the entire table to return the required results, but doing so can be inefficient. Indexes can reduce the effort required to locate results, but only if they are well designed.

SQL Server also uses indexes as part of its implementation of PRIMARY KEY and UNIQUE constraints. When you assign a PRIMARY KEY or UNIQUE constraint to a column or set of columns, SQL Server automatically indexes that column or set of columns. It does this to make it possible to quickly check whether a given value is already present.



A Useful Analogy

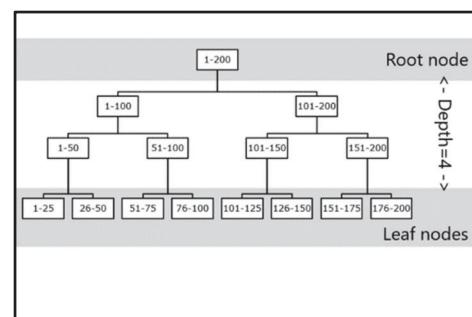
It is useful to consider an analogy that might be easier to relate to. Consider a physical library. Most libraries store books in a given order, which is basically an alphabetical order within a set of defined categories.

Note that, even when you store the books in alphabetical order, there are various ways to do it. The order of the books could be based on the title of the book or the name of the author. Whichever option is chosen makes one form of search easy and other searches harder. For example, if books were stored in title order, how would you find the ones that were written by a particular author? An index on a book's title and an index on the author would mean a librarian could find books quickly for either type of search.

Index Structures

Tree structures provide rapid search capabilities for large numbers of entries in a list.

Indexes in database systems are often based on binary tree structures. Binary trees are simple structures where at each level, a decision is made to navigate left or right. However, this style of tree can quickly become unbalanced and less useful; therefore, SQL Server uses a balanced tree.



Binary Tree Example

Using the topic slide as an example of a binary tree storing the values 1 to 200, consider how to find the value 136. If the data was stored randomly, each record would need to be examined until the desired value was found. It would take you a maximum 200 inspections to find the correct value.

Compare this against using an index. In the binary tree structure, 136 is compared against 100. As it is greater than 100, you inspect the next level down on the right side of the tree. Is 136 less than or greater than 150? It is less than, so you navigate down the left side. The desired value can be found in the page containing values 126 to 150, as 136 is greater than 125. Looking at each value in this page for 136, it is found at the 10th record. So a total of 13 values need to be compared, using a binary tree, against a possible 200 inspections against a random heap.

SQL Server indexes are based on a form of self-balancing tree. Whereas binary trees have, at most, two children per node, SQL Server indexes can have a larger number of children per node. This helps improve the efficiency of the indexes and reduces the overall depth of an index—depth being defined as the number of levels from the top node (called the root node) to the bottom nodes (called leaf nodes).

Selectivity, Density and Index Depth

Selectivity

Additional indexes on a table are most useful when they are highly selective. Selectivity is the most important consideration when selecting which columns should be included in an index.

For example, imagine how you would locate books by a specific author in a physical library by using a card file index. The process would involve the following steps:

- Finding the first entry for the author in the index.
- Locating the book in the bookcases, based on the information in the index entry.
- Returning to the index and finding the next entry for the author.
- Locating the book in the bookcases, based on the information in that next index entry.

You would need to keep repeating the same steps until you had found all of the books by that author. Now imagine doing the same for a range of authors, such as one-third of all of the authors in the library. You soon reach a point where it would be quicker to just scan the whole library and ignore the author index, rather than running backward and forward between the index and the bookcases.

Density

Density is a measure of the lack of uniqueness of the data in a table. It is a value between 0 and 1.0, and can be calculated for a column with the following formula:

$$\text{Density} = 1 / \text{number of unique values in a column}$$

A dense column is one that has a high number of duplicate values. An index will perform at its best when it has a low level of density.

- | |
|--|
| <ul style="list-style-type: none"> • Selectivity <ul style="list-style-type: none"> • A measure of how many rows are returned compared to the total number of rows • High selectivity means a small number of rows when related to the total number of rows
 • Density <ul style="list-style-type: none"> • A measure of the lack of uniqueness of data in the table • High density indicates a large number of duplicates
 • Index Depth <ul style="list-style-type: none"> • Number of levels within the index • Common misconception that indexes are deep |
|--|

Index depth

Index depth is a measure of the number of levels from the root node to the leaf nodes. Users often imagine that SQL Server indexes are quite deep, but the reality is different. The large number of children that each node in the index can have produces a very flat index structure. Indexes that are only three or four levels deep are very common.

Index Fragmentation

Index fragmentation is the inefficient use of pages within an index. Fragmentation can occur over time, as data in a table is modified.

For operations that read data, indexes perform best when each page of the index is as full as possible. Although indexes may initially start full, or relatively full, modifications to the data in the indexes can cause the need to split index pages.

From our physical library analogy, imagine a library that has full bookcases. What occurs when a new book needs to be added? If the book is added to the end of the library, the process is easy; however, if the book needs to be added in the middle of a full bookcase, there is a need to readjust all the surrounding bookcases.

- How does fragmentation occur?
 - SQL Server reorganizes index pages when data modifications cause index pages to split
- Types of fragmentation:
 - Internal – pages are not full
 - External – pages are out of logical sequence
- Detecting fragmentation
 - SQL Server Management Studio – Index Properties
 - System function – `sys.dm_db_index_physical_stats`

Internal vs. External Fragmentation

Internal fragmentation is similar to what would occur if an existing bookcase was split into two bookcases. Each bookcase would then be only half full.

External fragmentation relates to where the new bookcase would be physically located. It would probably need to be placed at the end of the library, even though it would “logically” need to be in a different order. This means that, to read the bookcases in order, you could no longer just walk directly from one bookcase to another. Instead, you would need to follow pointers around the library to track a chain between bookcases.

Detecting Fragmentation

SQL Server provides a measure of fragmentation in the `sys.dm_db_index_physical_stats` dynamic management view. The `avg_fragmentation_in_percent` column shows the percentage of fragmentation. SQL Server Management Studio also provides details of index fragmentation in the properties page for each index.

Demonstration: Viewing Index Fragmentation

In this demonstration, you will see how to:

- Identify fragmented indexes.
- View the fragmentation of an index in SSMS.

Demonstration Steps

Identify Fragmented Indexes

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Navigate to **D:\Demofiles\Mod05**, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**.
4. On the taskbar, click **Microsoft SQL Server Management Studio**.
5. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, and then click **Connect**.
6. On the **File** menu, point to **Open**, click **Project/Solution**.
7. In the **Open Project** dialog box, navigate to **D:\Demofiles\Mod05**, click **Demo05.ssmssln**, and then click **Open**.
8. In Solution Explorer, expand **Queries**, and then double-click **Demonstration 1.sql**.
9. Select the code under the **Step 1: Open a query window to the AdventureWorks database** comment, and then click **Execute**.
10. Select the code under the **Step 2: Query the index physical stats DMV** comment, and then click **Execute**.
11. Note the avg_fragmentation_in_percent returned.
12. Select the code under the **Step 4: Note that there are choices on the level of detail returned** comment, and then click **Execute**.
13. Select the code under the **Step 5: The final choice is DETAILED** comment, and then click **Execute**.

View the Fragmentation of an Index in SSMS

1. In Object Explorer, expand **Databases**, expand **AdventureWorks**, expand **Tables**, expand **Production.Product**, and then expand **Indexes**.
2. Right-click **AK_Product_Name (Unique, Non-Clustered)**, and then click **Properties**.
3. In the **Index Properties - AK_Product_Name** dialog box, in the **Select a page** pane, click **Fragmentation**.
4. Note the **Total fragmentation** is **75%**, and that this matches the results from the query executed in the previous task step 11.
5. In the **Index Properties - AK_Product_Name** dialog box, click **Cancel**.
6. Keep SQL Server Management Studio open for the next demonstration.

Categorize Activity

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

Items	
1	A factor of the number of rows returned, compared to the total number of rows in the index.
2	How unique the data is, compared to the other data in the index.
3	The number of unique levels between the root node and leaf nodes in the index.
4	The most important consideration when designing an index.

Category 1	Category 2	Category 3
Selectivity	Density	Depth

Lesson 2

Data Types and Indexes

Not all data types work equally well when included in an index. The size of the data and the selectivity of the search are the most important considerations for performance, but you should also consider usability. In this lesson, you will gain a better understanding of the impacts of the different data types on index performance.

Lesson Objectives

After completing this lesson, you will be able to:

- Discuss the impact of numeric data on indexes.
- Discuss the impact of character data on indexes.
- Discuss the impact of date-related data on indexes.
- Discuss the impact of globally-unique identifier (GUID) data on indexes.
- Discuss the impact of BIT data on indexes.
- Describe the benefits of using computed columns with indexes.

Numeric Index Data

When numeric values are used as components in indexes, a large number of entries can fit in a small number of index pages. This makes reading indexes based on numeric values very fast.

The type of numeric data will have an impact on the indexes' overall size and performance.

- | |
|--|
| • Using numeric values in indexes |
| • Benefits |
| • Small in size |
| • More values can fit in a single page |
| • Faster to read |
| • Negatives |
| • Small data types will be more dense |

Data Type	Storage Space
tinyint	1 byte
smallint	2 bytes
int	4 bytes
bigint	8 bytes
decimal(p,s) numeric(p,s)	5 to 17 bytes
smallmoney	4 bytes
money	8 bytes
real	4 bytes
float(n)	4 bytes or 8 bytes

As each page in an index is 8 KB in size, an index with only an int data type will hold a maximum of 2,048 values in a single page.

The disadvantage of using smaller numerical data types is that, inherently, the column will be more dense. As the range of numbers reduces, the number of duplicates increases.

Character Index Data

Character-based indexes are typically less efficient than numeric indexes, but character data is often used to search for a record—so, in those circumstances, an index can be very beneficial.

Character data values tend to be larger than numeric values. For example, a character column might hold a customer's name or address details. This means that far fewer entries can exist in a given number of index pages, which makes character-based indexes slower to seek.

Character-based indexes also tend to cause fragmentation problems because new values are seldom at the end of an index.

Data Type	Storage Space
char	8,000 bytes
varchar	8,000 bytes
text	2,147,483,647 bytes

The preceding table shows the maximum size for each of these columns. As character columns like varchar will only be as big as the largest data stored in them, these sizes could be considerably smaller.

- Character data types in indexes
- Benefits
 - Character data is often searched
 - Better performance than a heap
- Negatives
 - Slower to search than a numeric index
 - Can become fragmented because data does not tend to be sequential

Date-Related Index Data

Date-related data types are only slightly less efficient than integer data types. Date-related data types are relatively small, so can be compared and sorted quickly.

Data Type	Storage Space
date	3 bytes
datetime	8 bytes
datetimeoffset	10 bytes
smalldatetime	4 bytes

- Using date data types in indexes
- Benefits
 - Smaller in size
 - More values can fit in a single page
 - Quite faster to read
- Negatives
 - Small data types will be more dense

Data Type	Storage Space
time	5 bytes
timestamp	8 bytes

Like numerical data types, the column will inherently be more dense. As the range of dates reduces, the number of duplicates increases.

GUID Index Data

Globally unique identifier (GUID) values are reasonably efficient within indexes. There is a common misconception that they are large, but they are 16 bytes long and can be compared in a binary fashion. This means that they pack quite tightly into indexes and can be compared and sorted quickly.

Data Type	Storage Space
uniqueidentifier	16 bytes

GUIDs are typically used as keys in a table, so therefore each value is unique—this data type is very selective and has little density.

The downside to this uniqueness is that it will take longer to perform updates and deletes to records in the middle of the table. This is because the index may need to be updated and reordered.

- Using the GUID data type in indexes
- Benefits
 - Highly selective
 - Fast to read
- Negatives
 - Updates and deletes do not perform as well

BIT Index Data

There is a very common misconception that bit columns are not useful in indexes. This stems from the fact that there are only two values, which means a bit column is very dense.

Remember, though, that the selectivity of queries is the most important factor on the performance of an index. For example, consider a transaction table that contains 100 million rows, where one of the columns, IsFinalized, indicates whether a transaction has been completed. There might only be 500 transactions that are not completed. An index that uses the IsFinalized column would be very useful for finding the nonfinalized transactions. In this example, an index would be very useful, as it would be extremely selective.

Data Type	Storage Space
bit	1 bit

- Using BIT data type in indexes
- Benefits
 - Extremely small in size
 - More values can fit in a single page
 - Fast to read; in some circumstances could be highly selective
- Negatives
 - The index will be very dense

As the data type is extremely small, many more values can be stored in a page.

Indexing Computed Columns

A computed column is a column in a table that is derived from the values of other columns in the same table. For example, in a table that tracks product sales, you might create a computed column that multiplies the unit price of a product by the quantity of that product sold—to calculate a revenue value for each order. Applications that query the database could then obtain the revenue values without having to specify the calculation themselves.

When you create a computed column, SQL Server does not store the computed values; it only calculates them when the column is included in a query. Building an index on a computed column improves performance because the index includes the computed values, so SQL Server does not need to calculate them when the query is executed. Furthermore, the values in the index automatically update when the values in the base columns change, so the index remains up to date.

When you are deciding whether to index computed columns, you should consider the following:

- When the data in the base columns that the computed column references changes, the index is correspondingly updated. If the data changes frequently, these index updates can impair performance.
- When you rebuild an index on a computed column, SQL Server recalculates the values in the column. The amount of time that this takes will depend on the number of rows and the complexity of the calculation, but if you rebuild indexes often, you should consider the impact that this can have.
- You can only build indexes on computed columns that are deterministic.

- | |
|--|
| <ul style="list-style-type: none"> • Indexing computed columns • Benefits <ul style="list-style-type: none"> • Calculated values are stored in the index • Values updated automatically • Negatives <ul style="list-style-type: none"> • Frequent changes can impair performance • Computed columns must be deterministic |
|--|

Sequencing Activity

Put the following data types in order of the smallest to the maximum possible size.

Steps
BIT
date
bigint
datetimeoffset
uniqueidentifier
char
text

Lesson 3

Heaps, Clustered, and Nonclustered Indexes

Tables in SQL Server can be structured in two ways. Rows can be added in any order, or tables can be structured with rows added in a specific order. In this lesson, you will investigate both options, and gain an understanding of how each option affects common data operations.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the attributes of a table created without any order.
- Complete operations on a heap.
- Detail the characteristics of a clustered index, and its benefits over a heap.
- Complete operations on a clustered index.
- Describe how a primary key is different to a clustering key.
- Explain the reasons for using nonclustered indexes, and how they can be used in conjunction with heap and clustered indexes.
- Complete operations on a nonclustered index.

Heaps

The simplest table structure that is available in SQL Server is a heap. A heap is a table that has no enforced order for either the pages within the table, or for the data rows within each page. Data rows are added to the first available location within the table's pages that have sufficient space. If no space is available, additional pages are added to the table and the rows are placed in those pages.

Although no index structure exists for a heap, SQL Server tracks the available pages by using an entry in an internal structure called an Index Allocation Map (IAM).

Physical Library Analogy

In the physical library analogy, a heap would be represented by structuring your library so that every book is just placed in any available space in a bookcase. Without any other assistance, finding a book would involve scanning one bookcase after another—a bookcase being the equivalent of a page in SQL Server.

A heap is a table with:

- No specified order for pages within the table
- No specified order for rows within each page

Data will be inserted in the first space in a page that is found

Creation

To create a heap in SQL Server, all that is required is the creation of a table.

Creating a heap

```
CREATE TABLE Library.Book(
    BookID INT IDENTITY(1,1) NOT NULL,
    ISBN VARCHAR(14) NOT NULL,
    Title VARCHAR(4000) NOT NULL,
    AuthorID INT NOT NULL,
    PublisherID INT NOT NULL,
    ReleaseDate DATETIME NOT NULL,
    BookData XML NOT NULL
);
```

Physical Library Analogy

Now imagine that three additional indexes were created in the library, to make it easier to find books by author, International Standard Book Number (ISBN), and release date.

There was no order to the books on the bookcases, so when an entry was found in the ISBN index, the entry would refer to the physical location of the book. The entry would include an address like "Bookcase 12, Shelf 5, Book 3." That is, there would need to be a specific address for a book. An update to the book that required moving it to a different location would be problematic. One option for resolving this would be to locate all index entries for the book and update the new physical location.

An alternate option would be to leave a note in the location where the book used to be, pointing to where the book has been moved. In SQL Server, this is called a forwarding pointer—it means rows can be updated and moved without needing to update other indexes that point to them.

A further challenge arises if the book needs to be moved again. There are two ways in which this could be handled—another note could be left pointing to the new location or the original note could be modified to point to the new location. Either way, the original indexes would not need to be updated. SQL Server deals with this by updating the original forwarding pointer. This way, performance does not continue to degrade by having to follow a chain of forwarding pointers.

Remove Forwarding Pointers

When other indexes point to rows in a heap, data modification operations cause forwarding pointers to be inserted into the heap. Over time, this can cause performance issues.

Forwarding pointers were a common performance problem with tables in SQL Server that were structured as heaps. They can be resolved via the following command:

Forwarding pointers were a common performance problem with tables in SQL Server that were structured as heaps. They can be resolved via the following command:

Resolve forwarding pointer issues in a heap

```
ALTER TABLE Library.Book WITH REBUILD;
```

You can also use this command to change the compression settings for a table. Page and row compression are advanced topics that are beyond the scope of this course.

Operations on a Heap

The most common operations that are performed on tables are INSERT, UPDATE, DELETE, and SELECT. It is important to understand how each of these operations is affected by structuring a table as a heap.

Physical Library Analogy

In the library analogy, an INSERT operation would be executed by locating any gap that was large enough to hold the book and placing it there. If no space large enough is available, a new bookcase would be allocated and the book placed there. This would continue unless there was a limit on the number of bookcases that could fit in the library.

A DELETE operation could be imagined as scanning the bookcases until the book is found, removing the book, and throwing it away. More precisely, it would be like placing a tag on the book, to say that it should be thrown out the next time the library is cleaned up or space on the bookcase is needed.

An UPDATE operation would be represented by replacing a book with a (potentially) different copy of the same book. If the replacement book was the same (or smaller) size as the original book, it could be placed directly back in the same location as the original. However, if the replacement book was larger, the original book would be removed and the replacement moved to another location. The new location for the book could be in the same bookcase or in another bookcase.

There is a common misconception that including additional indexes always reduces the performance of data modification operations. However, it is clear that for the DELETE and UPDATE operations described above, having another way to find these rows might well be useful. In Module 6, you will see how to achieve this.

- **INSERT**
 - Each new row can be placed in the first available page with sufficient space
- **UPDATE**
 - The row can remain on the same page if it still fits; otherwise, it can be removed from the current page and placed on the first available page with sufficient space
- **DELETE**
 - Frees up space on the current page
 - Data is not overwritten, space is just flagged as available for reuse
- **SELECT**
 - Entire table needs to be scanned

Clustered Indexes

Rather than storing data rows of data as a heap, you can design tables that have an internal logical ordering. This kind of table is known as a clustered index or a rowstore.

A table that has a clustered index has a predefined order for rows within a page and for pages within the table. The order is based on a key that consists of one or more columns. The key is commonly called a clustering key.

The rows of a table can only be in a single order, so there can only be one clustered index on a table. An IAM entry is used to point to a clustered index.

- A clustered index:
 - Has pages that are logically ordered
 - Has rows that are logically ordered, and where possible, physically ordered within pages
 - Can only be declared once on a table
- The logical order is specified by a clustering key

There is a common misconception that pages in a clustered index are “physically stored in order.” Although this is possible in rare situations, it is not commonly the case. If it were true, fragmentation of clustered indexes would not exist. SQL Server tries to align physical and logical order while it creates an index, but disorder can arise as data is modified.

Index and data pages are linked within a logical hierarchy and also double-linked across all pages at the same level of the hierarchy—to assist when scanning across an index.

Creation

You can create clustered indexes, either directly by using the CREATE INDEX command, or automatically in situations where a PRIMARY KEY constraint is specified on the table:

Create an index directly on an existing table

```
CREATE INDEX IX_ISBN ON Library.Book (ISBN);
```

The following Transact-SQL will create a table. The alter statement then adds a constraint, with the side effect of a clustered index being created.

Create an index indirectly

```
CREATE TABLE Library.LogData
( LogID INT IDENTITY(1,1),
  LogData XML NOT NULL );

ALTER TABLE Library.LogData ADD CONSTRAINT PK_LogData PRIMARY KEY (LogId);
```

Updating

You can rebuild, reorganize and disable an index. The last option of disabling an index isn’t really applicable for clustered indexes, because disabling one doesn’t allow any access to the underlying data in the table. However, disabling a nonclustered index does have its uses. These will be discussed in a future topic.

Transact-SQL can be used to rebuild a single index.

Rebuild a specific index

```
ALTER INDEX IX_ISBN ON Library.Book REBUILD;
```

You can also rebuild all the indexes on a specified table.

Rebuild all indexes on a table

```
ALTER INDEX ALL ON Library.Book REBUILD;
```

The REORGANIZE statement can be used in the same way, either on a specific index or on a whole table.

Reorganize all indexes on a table

```
ALTER INDEX ALL ON Library.Book REORGANIZE;
```

Deleting

If a clustered index is created explicitly, then the following Transact-SQL will delete it:

Delete a clustered index

```
DROP INDEX IX_ISBN ON Library.Book;
```

A table will need to be altered to delete a clustered index, if it was created as a consequence of defining a constraint.

Delete a clustered index if created as part of adding a constraint

```
ALTER TABLE Library.LogData DROP CONSTRAINT PK_LogData;
```

Physical Library Analogy

In the library analogy, a clustered index is similar to storing all books in a specific order. An example of this would be to store books in ISBN order. Clearly, the library can only be sorted in one direction.

Operations on a Clustered Index

So far in this module, you have seen how common operations are performed on tables that are structured as heaps. It is important to understand how each of those operations is affected when you are structuring a table that has a clustered index.

Physical Library Analogy

In a library that is structured in ISBN order, an INSERT operation requires a new book to be placed in exactly the correct logical ISBN order. If there is space somewhere on the bookcase that is in the required position, the book can be placed into the correct location and all other books moved to accommodate the new one. If there is not sufficient space, the bookcase needs to be split. Note that a new bookcase would be physically placed at the end of the library, but would be logically inserted into the list of bookcases.

INSERT operations would be straightforward if the books were being added in ISBN order. New books could always be added to the end of the library and new bookcases added as required. In this case, no splitting is required.

When an UPDATE operation is performed, if the replacement book is the same size or smaller and the ISBN has not changed, the book can just be replaced in the same location. If the replacement book is larger, the ISBN has not changed, and there is spare space within the bookcase, all other books in the bookcase can slide along to enable the larger book to be replaced in the same spot.

If there was insufficient space in the bookcase to accommodate the larger book, the bookcase would need to be split. If the ISBN of the replacement book was different from the original book, the original book would need to be removed and the replacement book treated like the insertion of a new book.

A DELETE operation would involve the book being removed from the bookcase. (Again, more formally, it would be flagged as free in a free space map, but simply left in place for later removal.)

When a SELECT operation is performed, if the ISBN is known, the required book can be quickly located by efficiently searching the library. If a range of ISBNs was requested, the books would be located by finding the first book and continuing to collect books in order, until a book was encountered that was out of range, or until the end of the library was reached.

- INSERT
 - Each new row must be placed into the correct logical position
 - May involve splitting pages of the table
- UPDATE
 - The row can remain in the same place if it still fits and if the clustering key value is still the same
 - If the row no longer fits on the page, the page needs to be split
 - If the clustering key has changed, the row needs to be removed and placed in the correct logical position within the table
- DELETE
 - Frees up space by flagging the data as unused
- SELECT
 - Queries related to the clustering key can seek
 - Queries related to the clustering key can scan and avoid sorts

Primary Keys and Clustering Keys

As seen in a previous topic, you can create clustered indexes directly by using the CREATE INDEX command or as a side effect of creating a PRIMARY KEY constraint on the table.

It is very important to understand the distinction between a primary key and a clustering key. Many users confuse the two terms or attempt to use them interchangeably. A primary key is a constraint. It is a logical concept that is enforced by an index, but the index may or may not be a clustered index. When a PRIMARY KEY constraint is added to a table, the default action in SQL Server is to make it a clustered primary key—if no other clustered index already exists on the table.

You can override this action by specifying the word NONCLUSTERED when declaring the PRIMARY KEY constraint.

Creating a PRIMARY KEY without a clustered index

```
CREATE TABLE dbo.Author
(Name NVARCHAR(100) NOT NULL PRIMARY KEY NONCLUSTERED,
Publisher INT NOT NULL);
```

A primary key on a SQL table is used to uniquely identify rows in that table, and it must not contain any NULL values. In most situations, a primary key is a good candidate for a clustering key. However, a real world scenario, where the primary key may not be the clustered key, is in a table that requires a high volume of inserts. If this table has a sequential primary key, all inserts will be at the end of the table, in the last page. SQL Server may need to lock this page whilst inserting, forcing the inserts to become sequential instead of parallel.

Nonclustered Indexes

You have seen how tables can be structured as heaps or have clustered indexes. A third option is that you can create additional indexes on top of these tables to provide alternative ways to rapidly locate required data. These additional indexes are called nonclustered indexes.

A table can have up to 999 nonclustered indexes. Nonclustered indexes can be defined on a table—regardless of whether the table uses a clustered index or a heap—and are used to improve the performance of important queries.

Whenever you update key columns from the nonclustered index or update clustering keys on the base table, the nonclustered indexes need to be updated, too. This affects the data modification performance of the system. Each additional index that is added to a table increases the work that SQL Server might need to perform when modifying the data.

- Primary key
 - Must be unique
 - Cannot contain NULL values
 - Only one per table
 - Implemented as a constraint
- Clustering key
 - Must be unique
 - Specifies the logical ordering of rows
 - Only one per table
 - Can be automatically created

A nonclustered index :

- Can be on a heap or clustered index
- Will take up extra space
- Needs to be updated when the underlying data is modified

A table can have a maximum of 999 nonclustered indexes.

rows in the table. You must take care to balance the number of indexes that are created against the overhead that they introduce.

Creation

Similar to clustered indexes, nonclustered indexes are created explicitly on a table. The columns to be included also need to be specified.

Creating a nonclustered index

```
CREATE NONCLUSTERED INDEX IX_Book_Publisher  
    ON Library.Book (PublisherID, ReleaseDate DESC);
```

There is an option that is unique to nonclustered indexes. They can have an additional INCLUDE option on declaration that is used to create covering indexes. These will be discussed in further detail in Module 6: *Advanced Indexing*.

Creating a covering nonclustered index

```
CREATE NONCLUSTERED INDEX NCIX_Author_Publisher  
    ON Library.Book (BookID)  
    INCLUDE (AuthorID, PublisherID, ReleaseDate);
```

Updating

The Transact-SQL for nonclustered indexes is exactly the same as for clustered indexes. You can rebuild, reorganize and disable an index.

Disabling an index can be very useful for nonclustered indexes on tables that are going to have large amounts of data, either inserted or deleted. Before performing these data operations, all nonclustered indexes can be disabled. After the data has been processed, the indexes can then be enabled by executing a REBUILD statement. This reduces the performance impacts of having nonclustered indexes on tables.

Deletion

The same Transact-SQL that is used for clustered indexes will delete a nonclustered index.

Delete a nonclustered index

```
DROP INDEX NCIX_Author_Publisher ON Library.Book;
```

Physical Analogy

The nonclustered indexes can be thought of as indexes that point back to the bookcases. They provide alternate ways to look up the information in the library. For example, they might give access by author, by release date, or by publisher. They can also be composite indexes where you might find an index by release date, within the entries for each author. Composite indexes will be discussed in the next lesson.

Operations on Nonclustered Indexes

The operations of a nonclustered index are dependent on the underlying table structure they are declared on. However, there are common considerations across both structures, the most important being that every extra nonclustered index is taking more space within the database. Another downside is that, when data is modified in the underlying table, each nonclustered index will need to be kept up to date.

Due to the downsides of nonclustered indexes, if large batches of inserts or deletes are being performed, it is common practice to disable nonclustered indexes before the data operations, and then recreate them.

- **INSERT**
 - Each nonclustered index that is added to a table will decrease the performance of inserts
- **UPDATE**
 - The index will need to be kept up to date if the location of the data changes
- **DELETE**
 - Similar to updates, deleted data needs to be removed from the index
- **SELECT**
 - Performance improvements for queries that the index covers

Physical Analogy

It does not matter how the library is structured—whether the books are stored in ISBN order, or category and author order, or randomly, a nonclustered index is like an extra card index pointing to the locations of books in the bookcases. These extra indexes can be on any attribute of a book; for example, the release date, or whether it has a soft or hard cover. The cards in the index will have a pointer to the book's physical location on a shelf.

Each of these extra indexes then need to be maintained by the librarian. When a librarian inserts a new book in the library, they should make a note of its location in each card index. This is true for removing a book, or updating its location to another shelf. Each of these operations requires updates to be made to every card index.

These extra indexes have an advantage when they are used to search for books. The additional indexes will improve the performance of finding books released in 2003, for example. Without this extra index, the librarian would have to check the release date of every single book, to see if it matched the required date.

Demonstration: Working with Clustered and Nonclustered Indexes

In this demonstration, you will see how to:

- Create a clustered index.
- Remove fragmentation on a clustered index.
- Create a covering index.

Demonstration Steps

1. In SQL Server Management Studio, in Solution Explorer, double-click **Demonstration 2.sql**.
2. Select the code under the **Step 1: Open a new query window against the tempdb database** comment, and then click **Execute**.
3. Select the code under the **Step 2: Create a table with a primary key specified** comment, and then click **Execute**.
4. Select the code under the **Step 3: Query sys.indexes to view the structure** comment, and then click **Execute**.

5. In Object Explorer, expand **Databases**, expand **System Databases**, expand **tempdb**, expand **Tables**, expand **dbo.PhoneLog**, and then expand **Indexes**.
6. Note that a clustered index was automatically created on the table.
7. Select the code under the **Step 4: Insert some data into the table** comment, and then click **Execute**.
8. Select the code under the **Step 5: Check the level of fragmentation via sys.dm_db_index_physical_stats** comment, and then click **Execute**.
9. Scroll to the right, and note the **avg_fragmentation_in_percent** and **avg_page_space_used_in_percent** returned.
10. Select the code under the **Step 7: Modify the data in the table - this will increase data and cause page fragmentation** comment, and then click **Execute**.
11. Select the code under the **Step 8: Check the level of fragmentation via sys.dm_db_index_physical_stats** comment, and then click **Execute**.
12. Scroll to the right, and note the **avg_fragmentation_in_percent** and **avg_page_space_used_in_percent** returned.
13. Select the code under the **Step 10: Rebuild the table and its indexes** comment, and then click **Execute**.
14. Select the code under the **Step 11: Check the level of fragmentation via sys.dm_db_index_physical_stats** comment, and then click **Execute**.
15. Scroll to the right, and note the **avg_fragmentation_in_percent** and **avg_page_space_used_in_percent** returned.
16. On the **Query** menu, click **Include Actual Execution Plan**.
17. Select the code under the **Step 13: Run a query showing the execution plan** comment, and then click **Execute**.
18. Review the execution plan.
19. Select the code under the **Step 14: Create a covering index, point out the columns included** comment, and then click **Execute**.
20. Select the code under the **Step 15: Run the query showing the execution plan (CTR+M) – it now uses the new index** comment, and then click **Execute**.
21. Review the execution plan.
22. Select the code under the **Step 16: Drop the table** comment, and then click **Execute**.
23. Keep SQL Server Management Studio open for the next demonstration.

Categorize Activity

Categorize each attribute of an index. Indicate your answer by writing the attribute number to the right of each index.

Items	
1	Data is stored in the table wherever there is space.
2	Data is stored by a key in a specified order.
3	Will be defined on top of a heap or rowstore.
4	Most efficient operation is an insert.
5	Can improve the performance of updates, deletes and selects.
6	Can improve the performance of selects.
7	Best used when scanning for data.
8	Best used when seeking for data.

Category 1	Category 2	Category 3
Heap	Clustered	Nonclustered

Lesson 4

Single Column and Composite Indexes

The indexes discussed so far have been based on data from single columns. Indexes can also be based on data from multiple columns, and constructed in ascending or descending order. This lesson investigates these concepts and the effects that they have on index design, along with details of how SQL Server maintains statistics on the data that is contained within indexes.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the differences between single column and composite indexes.
- Describe the differences between ascending and descending indexes.
- Explain how SQL Server keeps statistics on indexes.

Single Column vs. Composite Indexes

Indexes can be constructed on multiple columns rather than on single columns. Multicolumn indexes are known as composite indexes.

In applications, composite indexes are often more useful than single column indexes. The advantages of composite indexes are:

- Higher selectivity.
- The possibility of avoiding the need to sort the output rows.

- Indexes are not always constructed on a single column
- Composite indexes tend to be more useful than single column indexes:
 - Having an index sorted first by customer, then by order date, makes it easy to find orders for a particular customer on a particular date
 - Two columns together might be selective while neither is selective on its own
- Index on A,B is not the same as an index on B,A

In our physical library analogy, consider a query that required the location of books by a publisher within a specific release year. Although a publisher index would be useful for finding all of the books that the publisher released, it would not help to narrow down the search to those books within a specific year. Separate indexes on the publisher and the release year would not be useful, but an index that contained both publisher and release year could be very selective.

Similarly, an index by topic would be of limited value. After the correct topic had been located, it would be necessary to search all of the books on that topic to determine if they were by the specified author.

The best option would be an author index that also included details of each book's topic. In that case, a scan of the index pages for the author would be all that was required to work out which books needed to be accessed.

When you are constructing composite indexes, in the absence of any other design criteria, you should typically index the most selective column first. The order of columns in a composite index is important, not only for performance, but also for whether the query optimizer will even use the index. For example, an index on City, State would not be used in queries where State is the only column in the WHERE clause.

Considerations

The following should also be considered when choosing columns to add to a composite index:

- Is the column selective? Only columns that are selective should be used.
- How volatile is the column? Columns that are frequently updated will likely cause an index to be rebuilt. Choose columns that have more static data.
- Is the column queried upon? This column should be included providing it passes the above considerations.
- The most selective columns should be first in the composite index; columns with inequality predicates should be towards the end.
- Keep the number of columns to a minimum, as each column added increases the overall size of the composite index.

A specific type of composite index is a covering index. This kind of column is outside the scope of this module but is covered in Module 6: *Advanced Indexing*.

Ascending vs. Descending Indexes

Each component of an index can be created in an ascending or descending order. By default, if no order is specified when creating an index, then it will be in ascending order. For single column indexes, ascending and descending indexes are equally useful. For composite indexes, specifying the order of individual columns within the index might be useful.

In general, it makes no difference whether a single column index is ascending or descending. From our physical library analogy, you could scan either the bookshelves or the indexes from either end.

The same amount of effort would be required, no matter which end you started from.

Composite indexes can benefit from each component having a different order. Often this is used to avoid sorts. For example, you might need to output orders by date descending within customer ascending. From our physical library analogy, imagine that an author index contains a list of books by release date within the author index. Answering the query would be easier if the index was already structured this way.

- Indexes could be constructed in ascending or descending order
- In general, for single column indexes, both are equally useful
 - Each layer of a SQL Server index is double-linked (that is, linked in both directions)
 - SQL Server can start at either end and work towards the other end
- Each component of a composite index can be ascending or descending
 - Might be useful for avoiding sort operations

Index Statistics

SQL Server keeps statistics on indexes to assist when making decisions about how to access the data in a table.

By default, these statistics are automatically created and updated on indexes.

Earlier in this module, you saw that SQL Server needs to make decisions about how to access the data in a table. For each table that is referenced in a query, SQL Server might decide to read the data pages or it may decide to use an index.

It is important to realize that SQL Server must make this decision before it begins to execute a query. This means that it needs to have information that will assist it in making this determination. For each index, SQL Server keeps statistics that tell it how the data is distributed. The query optimizer uses these statistics to estimate the cardinality, or number of rows, that will be in the query result.

- SQL Server needs to have knowledge of the layout of the data in a table or index **before** it optimizes and executes queries
 - Needs to create a reasonable plan for executing the query
 - Important to know the usefulness of each index
 - Selectivity is the most important metric
- By default, SQL Server automatically creates statistics on indexes
 - Can be disabled
 - Recommendation is to leave auto-creation and auto-update enabled

Identifying Out of Date Statistics

As data in a table is updated, deleted or inserted, the associated statistics can become out of date. There are two ways to explore the accuracy of statistics for any given table:

1. Inspect a queries execution plan and check that the "Actual Number of Rows" and "Estimated Number of Rows" are approximately the same.
2. Use a Transact-SQL command—DBCC SHOW_STATISTICS—and check the Updated column.

If statistics are determined to be out of date they can be manually updated with one of the following Transact-SQL commands:

Update statistics

```
/* Update all statistics in a database */
EXEC sp_updatestats;

/* Update all the statistics on a specific table */
UPDATE STATISTICS Production.Product;

/* Update the statistics on a specific index */
UPDATE STATISTICS Production.Product PK_Product_ProductID;
```

Physical Library Analogy

When discussing the physical library analogy earlier, it was mentioned that, if you were looking up the books for an author, it could be useful to use an index that is ordered by author. However, if you were locating books for a range of authors, there would be a point at which scanning the entire library would be quicker than running backward and forward between the index and the bookshelves.

The key issue here is that, before executing the query, you need to know how selective (and therefore useful) the indexes would be. The statistics that SQL Server holds on indexes provide this knowledge.

Demonstration: Viewing Index Statistics

In this demonstration, you will see how to:

- View the statistics of an index via Transact-SQL.
- View the statistics of an index with SSMS.
- View the database settings related to statistics.

Demonstration Steps

Use Transact-SQL to View Statistics

1. In SQL Server Management Studio, in Solution Explorer, double-click **Demonstration 3.sql**.
2. Select the code under the comment **Step 1: Run the Transact-SQL up to the end step 1**, and then click **Execute**.
3. Walk through the important columns in the results.
4. On the **Query** menu, click **Include Actual Execution Plan**.
5. Select the code under the comment **Step 2: Check the freshness of the statistics, CTRL-M to switch on the Execution Plan**, and then click **Execute**.
6. In the Execution Plan pane, scroll right and point at the last Clustered Index Scan. Note that the actual and estimated number of rows are equal.

Use SQL Server Management Studio to View Statistics

1. In Object Explorer, expand **Databases**, expand **AdventureWorks**, expand **Tables**, expand **HumanResources.Employee**, and then expand **Statistics**.
2. Right-click **AK_Employee_LoginID**, and then click **Properties**.
3. In the **Statistics Properties - AK_Employee_LoginID** dialog box, in the **Select a page** section, click **Details**.
4. Review the details, and then click **Cancel**.

Inspect the Statistics Settings for a Database

1. In Object Explorer, under **Databases**, right-click **AdventureWorks**, and then click **Properties**.
2. In the **Database Properties - AdventureWorks** dialog box, in the **Select a page** section, click **Options**.
3. In the **Other options** list, scroll to the top, under the **Automatic** heading, note that the **Auto Create Statistics** and **Auto Update Statistics** are set to **True**.
4. In the **Database Properties - AdventureWorks** dialog box, click **Cancel**.
5. Close SSMS without saving any changes.

Lab: Implementing Indexes

Scenario

One of the most important decisions when designing a table is to choose an appropriate table structure. In this lab, you will choose an appropriate structure for some new tables required for the relationship management system.

Objectives

After completing this lab, you will be able to:

- Create a table without any indexes.
- Create a table with a clustered index.
- Add a nonclustered key in the form of a covering index.

Estimated Time: 30 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Creating a Heap

Scenario

The design documentation requires you to create tables to store sales related data. You will create these two tables to support the requirement of the sales department.

The supporting documentation for this exercise is located in **D:\Labfiles\Lab05\Starter\Supporting Documentation.docx**.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Review the Documentation
3. Create the Tables

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab05\Starter** folder as Administrator.

► Task 2: Review the Documentation

- Review the requirements in **Supporting Documentation.docx** in the **D:\Labfiles\Lab05\Starter** folder and decide how you are going to meet them.

► Task 3: Create the Tables

1. Create a table based on the supporting documentation for Table 1: Sales.MediaOutlet.
2. Create a table based on the supporting documentation for Table 2: Sales.PrintMediaPlacement.

Results: After completing this exercise, you will have created two new tables in the AdventureWorks database.

Exercise 2: Creating a Clustered Index

Scenario

The sales department has started to use the new tables, and are finding that, when trying to query the data, the performance is unacceptable. They have asked you to make any database changes you can to improve performance.

The main tasks for this exercise are as follows:

1. Add a Clustered Index to Sales.MediaOutlet
2. Add a Clustered Index to Sales.PrintMediaPlacement

► Task 1: Add a Clustered Index to Sales.MediaOutlet

1. Consider which column is best suited to an index.
2. Using Transact-SQL statements, add a clustered index to that column on the **Sales.MediaOutlet** table.

Consider implementing the index by creating a unique constraint.

3. Use Object Explorer to check that the index was created successfully.

► Task 2: Add a Clustered Index to Sales.PrintMediaPlacement

1. Consider which column is best suited to an index.
2. Using Transact-SQL statements, add a clustered index to that column on the **Sales.PrintMediaPlacement** table.
3. Use Object Explorer to check that the index was created successfully.

Results: After completing this exercise, you will have created clustered indexes on the new tables.

Exercise 3: Creating a Covering Index

Scenario

The sales team has found that the performance improvements that you have made are not working for one specific query. You have been tasked with adding additional performance improvements to handle this query.

The main tasks for this exercise are as follows:

1. Add Some Test Data
2. Run the Poor Performing Query
3. Create a Covering Index
4. Check the Performance of the Sales Query

► Task 1: Add Some Test Data

- Run the Transact-SQL in **D:\Labfiles\Lab05\Starter\InsertDummyData.sql** to insert test data into the two tables.

► **Task 2: Run the Poor Performing Query**

1. Switch on Include Actual Execution Plan.
2. Run the Transact-SQL in **D:\Labfiles\Lab05\Starter\SalesQuery.sql**.
3. Examine the Execution Plan.
4. Note the missing index warning in SQL Server Management Studio.

► **Task 3: Create a Covering Index**

1. On the **Execution Plan** tab, right-click the green **Missing Index** text and click **Missing Index Details**.
2. Use the generated Transact-SQL to create the missing covering index.
3. Use Object Explorer to check that the index was created successfully.

► **Task 4: Check the Performance of the Sales Query**

1. Rerun the sales query.
2. Check the Execution Plan and ensure the database engine is using the new **NCI_PrintMediaPlacement** index.
3. Close SQL Server Management Studio without saving any changes.

Results: After completing this exercise, you will have created a covering index suggested by SQL Server Management Studio.

Module Review and Takeaways



- Best Practice:** Choose columns with a high level of selectivity for indexes.
- Rebuild highly fragmented indexes.
- Use nonclustered indexes to improve the performance of particular queries, but balance their use with the overhead that they introduce.
- Use actual execution plans to obtain missing index hints.

Module 6

Designing Optimized Index Strategies

Contents:

Module Overview	6-1
Lesson 1: Index Strategies	6-2
Lesson 2: Managing Indexes	6-7
Lesson 3: Execution Plans	6-16
Lesson 4: The Database Engine Tuning Advisor	6-25
Lesson 5: Query Store	6-27
Lab: Optimizing Indexes	6-34
Module Review and Takeaways	6-37

Module Overview

Indexes play an important role in enabling SQL Server to retrieve data from a database quickly and efficiently. This module discusses advanced index topics including covering indexes, the INCLUDE clause, query hints, padding and fill factor, statistics, using DMOs, the Database Tuning Advisor, and Query Store.

Objectives

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

- What a covering index is, and when to use one.
- The issues involved in managing indexes.
- Actual and estimated execution plans.
- How to use Database Tuning Advisor to improve the performance of queries.
- How to use Query Store to improve query performance.

Lesson 1

Index Strategies

This lesson considers index strategies, including covering indexes, the INCLUDE clause, heaps and clustered indexes, and filtered indexes.

Lesson Objectives

After completing this lesson, you will be able to:

- Understand when to use a covering index.
- Explain when to use the INCLUDE clause.
- Understand the performance difference between a heap and a clustered index.
- Understand when to use a filtered index.

Covering Indexes

Covering Indexes Include All Columns

A covering index is a nonclustered index that includes all the columns required by a specific query. Because the query has all the columns in the nonclustered index, there is no need for SQL Server to retrieve data from clustered indexes. This can dramatically improve the performance of a query. When an index includes all the columns required by a query, the index is said to cover the query.

- A covering index includes all the columns returned by a query
 - No need for the query optimizer to use clustered indexes
- Use covering indexes for:
 - Frequently used queries
 - Poorly performing queries
- When the query is no longer needed, drop the covering index

When Should You Use a Covering Index?

You can use a covering index to improve the performance of any query that is running too slowly. Although indexes improve query performance, they also create an overhead. When data is entered or modified, related indexes have to be updated. So adding a new index is always a balance between decreasing the time taken to retrieve data, against the time taken to add or change data. Create covering indexes where they can be most effective—for queries that have performance problems, or for queries that are used frequently.

Drop Unnecessary Indexes

An index is a covering index when it includes all the columns required by the query. However, when that query is no longer required, the index has no further purpose and should be dropped. Keeping unnecessary indexes makes data entry and modification slower, and statistics take longer to maintain. Dropping unnecessary indexes helps to keep your database running smoothly.

 **Note:** A covering index is just a nonclustered index that includes all the columns required by a particular query. It is sometimes referred to as an "index covering the query".

Using the INCLUDE Clause

Index Limitations

We have already seen that a covering index can improve the performance of some queries; however, SQL Server has limits on how large an index can be. In SQL Server, these limitations are:

- Maximum of 32 columns.
- Clustered indexes must be 900 bytes or less.
- Nonclustered indexes must be 1700 bytes or less.
- Indexes cannot include LOB columns and key columns. LOB data types include ntext, text, varchar(max), nvarchar(max), varbinary(max), xml, and image.
- All columns must be from the same table or view.

- SQL Server indexes have a number of limitations
- Columns added with the INCLUDE clause are nonkey columns
- The INCLUDE clause adds columns at the leaf level of an index
- Use INCLUDE when:
 - You would exceed the number of columns or max size for an index
 - You want to create a covering index
 - You want to add columns with larger data types that are only used in the SELECT statement
- Nonkey column data is stored twice

Use the INCLUDE clause if you want to create a covering index, but some columns are of excluded data types, or the index might already be at its maximum size.

Key and Nonkey Columns

Columns in an index are known as key columns, whereas columns added using the INCLUDE clause are known as nonkey columns. You can add nonkey columns to any index using the INCLUDE clause, but they give the most benefit when they cover a query.

Nonkey Columns Are at the Leaf Level

Columns added using the INCLUDE clause are added at the leaf level of the index, rather than in the higher nodes. This makes them suitable for SELECT statement columns, rather than columns used for joins or sorting. Use the INCLUDE clause to make an index more efficient, keeping smaller columns used for lookups as key columns, and adding larger columns as nonkey columns.



Note: All columns in an index must be from the same table. If you want an index with columns from more than one table, create an indexed view.

When to Use the INCLUDE Clause

A nonclustered index may include two table columns—you can use the INCLUDE clause to add three more columns, and create a covering index. In this example, key columns refer to the two columns of the nonclustered index; columns added using the INCLUDE clauses are the nonkey columns.

The most efficient way of using the INCLUDE clause is to use it for columns with larger data types that are only used in the SELECT statement. Make columns that are used in the WHERE or GROUP BY clauses into key columns, so keeping indexes small and efficient.

The following index is created with columns used for searching and sorting as key columns, with the larger columns that only appear in the SELECT statement as INCLUDED columns:

Covering Index Using the INCLUDE Clause

```
USE AdventureWorks2016;
GO

CREATE NONCLUSTERED INDEX AK_Product
    ON Production.Product
    (ProductID ASC, ListPrice ASC)
    INCLUDE (Name, ProductNumber, Color)
GO
```

Data Stored Twice

Adding columns with larger data types to an index using the INCLUDE clause means that the data is stored both in the original table, and in the index. So whilst covering indexes can give a performance boost, this strategy increases the required disk space. There is also an overhead when inserting, updating, and deleting data as the index is maintained.

For more information about the INCLUDE clause, see Microsoft Docs:

Create Indexes with Included Columns

<http://aka.ms/K39f9y>

Heap vs. Clustered Index

What Is a Heap?

A heap is a table that has been created without a clustered index, or a table that had a clustered index that has now been dropped. A heap is unordered: data is written to the table in the order in which it is created. However, you cannot rely on data being stored in the same order as it was created, because SQL Server can reorder the data to store it more efficiently.

- A heap is a table without a clustered index
 - Rarely used
 - Perhaps for very small tables
 - Perhaps for tables that require the data writing to disk very quickly
- A clustered index defines the physical sequence of table data
 - Only one clustered index per table
 - Often the primary key
 - Tables normally have a clustered index

 **Note:** You cannot rely on any data set being in a particular order unless you use the ORDER BY clause—whether or not the table is a heap. Although data might appear to be in the order you require, SQL Server does not guarantee that data will always be returned in a particular sequence, unless you specify the order.

Tables are normally created with a primary key, which automatically creates a clustered index. Additional nonclustered indexes can then be created as required. However, you can also create nonclustered indexes on a heap. So why would you want to create a heap?

When to Use a Heap

There are two main reasons to use a heap:

1. The table is so small that it doesn't matter. A parameters table, or lookup table that contains only a few rows, might be stored as a heap.
2. You need to write data to disk as fast as possible. A table that holds log records, for example, might need to write data without delay.

However, it is fair to say that effective use of heaps is rare. Tables are almost always more efficient when created with a clustered index because, with a heap, the whole table must be scanned to find a record.

 **Note:** You can make the storage of a heap more efficient by creating a clustered index on the table, and then dropping it. Be aware, however, that each time a clustered index is created or dropped from a table, the whole table is rewritten to disk. This can be time-consuming, and requires there to be enough disk space in **tempdb**.

For more information about heaps, see Microsoft Docs:

 **Heaps (Tables without Clustered Indexes)**

<http://aka.ms/Dqip6i>

What Is a Clustered Index?

A clustered index defines the order in which the physical table data is stored. A table can have only one clustered index because the data can only be stored in one sequence. A clustered index is automatically created when a primary key is created, providing there is not already a clustered index on the table. Tables in a relational database almost always have a clustered index. A well-designed clustered index improves query performance, and uses less system resources.

Filtered Index

What Is a Filtered Index?

A filtered index is created on a subset of the table's records. The index must be a nonclustered index, and the filter defined using a WHERE clause. Filtered indexes improve query performance when queries use only some of a table's data, and that subset can be clearly defined.

- A filtered index is an index on a subset of a table
- Suitable for queries that access a defined subset of data, such as:
 - Sales for the northern region
 - Only finished goods
- Filtered indexes can only be created on nonclustered indexes
 - Not clustered indexes
 - Not views

When to Use a Filtered Index

Filtered indexes can be used when a table contains subsets of data that are queried frequently. For example:

- Nullable columns that contain few non-NULL values.
- Ranges of values that are queried frequently, such as financial values, dates, or geographic regions.
- Category data, such as status codes.

Consider a products table that is frequently queried for products that have the FinishedGoodsFlag. Although the table holds both components and finished items, the marketing department almost always queries the table for finished items. Creating a filtered index will improve performance, especially if the table is large.

The following code example shows an index filtered on finished goods only:

Filtered Index

```
USE AdventureWorks2016;
GO

CREATE INDEX ix_product_finished
    ON Production.Product (FinishedGoodsFlag)
    WHERE FinishedGoodsFlag = 1;
```

Creating filtered indexes not only increases the performance of some queries, but also reduces the size of an index, taking up less space on disk and making index maintenance operations faster.

Filtered Index Limitations

However, there are limitations to creating filtered indexes. These are:

- Filtered indexes can only be created on tables, not views.
- Filter predicates support only simple comparisons.
- If you need to perform a data conversion, this can only be done on the right-hand side of the predicate operator.

For more information about filtered indexes, see Microsoft Docs:



[Create Filtered Indexes](http://aka.ms/mudq8y)

<http://aka.ms/mudq8y>

Check Your Knowledge

Question
Your sales table holds data for customers across the world. You want to improve the performance of a query run by the French office that calculates the total sales, only for customers based in France. Which index would be most appropriate, and why?
Select the correct answer.
A covering index.
A clustered index.
A filtered index.
A nonclustered index on a heap.
Add region as a nonkey column using the INCLUDE clause.

Lesson 2

Managing Indexes

This lesson introduces topics related to managing indexes, including fill factor, padding, statistics, and query hints.

Lesson Objectives

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

- The FILL FACTOR setting, and how to use it.
- The PAD INDEX option.
- What statistics are, and how to manage them.
- When to use query hints.

What Is Fill Factor?

Fill factor is a setting that determines how much spare space remains on each leaf-level page of an index and is used to reduce index fragmentation. As indexes are created and rebuilt, the leaf-level pages are filled to a certain level—determined by the fill factor—and the remaining space is left for future growth. You can configure a default fill factor for the server, or for each individual index.

Fill Factor Settings

The fill factor sets the amount of spare space that will be left on each leaf-level page as it fills with data. The fill factor is expressed as the percentage of the page that is filled with data—a fill factor of 75 means that 75 percent of the page will be filled with data, and 25 percent will be left spare. Fill factors of 0 and 100 are treated as the same, with the leaf-level page being completely filled with data.

- What is fill factor?
 - A setting that defines spare space on the leaf level of index pages
 - Set either for the server, or for individual indexes
 - A fill factor of 75 fills each leaf page up to 75 percent full
 - By leaving space, you reduce the need for page splits as new data is added
 - Provided data is added evenly between pages
 - You can set fill factor
 - For the SQL Server instance
 - For each index

How Fill Factor Improves Performance

New data is added to the appropriate leaf-level page. When a page is full, however, it must split before new data can be added. This means that an additional leaf-level page is created, and the data is split between the old and the new pages. Page splitting slows down data inserts and should be minimized. Set the fill factor according to how much data you expect to be added between index reorganizations or rebuilds.

 **Note:** Adding a fill factor less than 0 or 100 assumes that data will be added roughly evenly between leaf-level pages. For example, an index with a LastName key would have data added to different leaf-level pages. However, when data is always added to the last page, as in the case of an IDENTITY column, fill factor will not necessarily prevent page splits.

View the Default Fill Factor Setting

If you do not specify a fill factor for an index, SQL Server will use the default. Later in this module, you will see how to set the default, but you can view the default by running `sp_configure`.

Use Transact-SQL to view the default fill factor for the server:

Sp_configure

```
EXEC sp_configure 'show advanced options', '1';
RECONFIGURE;

sp_configure 'fill factor';
```



Note: Fill factor is an advanced option for `sp_configure`. This means you must set “show advanced options” to 1 before you can view the default fill factor settings, or change the settings.

What Is Pad Index?

Pad index is a setting used in conjunction with fill factor. It determines whether space is left on the intermediate nodes of an index.

The option `WITH PAD_INDEX` specifies that the same amount of space should be left at the intermediate nodes of an index, as is left on the leaf nodes of an index. Pad index uses the same percentage as specified with fill factor. However, SQL Server always leaves at least two rows on an intermediate index page, and will make sure there is enough space for one row to be added to an intermediate page.

`PAD_INDEX` works in conjunction with `FILL FACTOR`.

- `WITH PAD_INDEX` specifies that space should be left at the intermediate node-level pages of an index
- `PAD_INDEX` is used together with `FILL FACTOR`
- The fill factor percentage value is used by pad index
- `PAD_INDEX` is off by default

PAD_INDEX

```
USE AdventureWorks2016;
GO

CREATE INDEX ix_person_lastname
    ON Person.Person (LastName)
    WITH PAD_INDEX, FILLFACTOR = 10
```

Implementing Fill Factor and Padding

In the last two topics, we discussed fill factor and pad index when managing indexes. This topic covers how to implement the two settings.

Setting Default Fill Factor

If the fill factor is not specified for an index, SQL Server uses the default. There are two ways of setting the default—either by using the graphical user interface (GUI), or by using Transact-SQL. To set the default using the SSMS GUI, follow these steps:

- Set the default fill factor at the instance level
 - Using SSMS Object Explorer, right-click the instance name and select Database Properties
 - Type or select Fill Factor value
 - Or use Transact-SQL and `sp_configure`
- Using Transact-SQL, use `CREATE INDEX` or `ALTER INDEX` to set fill factor and pad index
- Index properties can be set using SSMS and Object Explorer
 - Expand the tree to the relevant index
 - Right-click and select Properties
 - Options > Storage > set fill factor and pad index

1. In Object Explorer, right-click the SQL Server instance name and select **Properties** from the menu.
2. From the **Server Properties** dialog box, select the **Database Settings** page.
3. The **Default index fill factor** is the first option. Set a value between 0 and 100 by typing, or selecting a value.
4. Click **OK** to save and close the dialog box.

Alternatively, you can amend the default fill factor using a Transact-SQL script. This has the same effect as setting the fill factor through the GUI—if you check in the server properties, you will see that the value has changed.

Setting the Default Fill Factor using Transact-SQL:

Sp_configure

```
sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO
sp_configure 'fill factor', 75;
GO
RECONFIGURE;
GO
```

Setting the Fill Factor for an Index

When you create a new index, you can set the fill factor. This will override the server default. You can specify the fill factor in a number of ways:

1. Use Transact-SQL to set the fill factor.
2. Use the SSMS GUI and index properties.

Create an index and specify the fill factor and pad index.

CREATE INDEX

```
USE AdventureWorks2016;
GO
-- Creates the IX_AddressID_City_PostalCode index
-- on the Person.Address table with a fill factor of 75.

CREATE INDEX IX_AddressID_City_PostalCode
    ON Person.Address
    (AddressID, City, PostalCode)
    WITH PAD_INDEX, FILLFACTOR = 80;
GO
```

Amend the fill factor and rebuild the index.

ALTER INDEX

```
USE AdventureWorks2016;
GO

-- Rebuild the IX_JobCandidate_BusinessEntityID index
-- with a fill factor of 90.

ALTER INDEX IX_JobCandidate_BusinessEntityID
    ON HumanResources.JobCandidate
    REBUILD WITH (FILLFACTOR = 90);
GO
```



Best Practice: Amending the fill factor will cause the index to be rebuilt. Make the change at a time when the database is not being used heavily, or when it is not being used at all.

Index Properties

To change the fill factor using index properties:

1. In Object Explorer, expand the **Databases** node, in the database you are working on, then expand the **Tables** node. Expand the table relating to the index, and expand **Indexes**.
2. Right-click the index name, and select **Properties**.
3. Select the **Options** page, and find the **Storage** section.
4. For **Fill Factor**, enter a value between 0 and 100.
5. **Pad index** may be set to True or False.
6. Click **OK** to save and close the **Properties** dialog box.



Note: Setting a very low fill factor increases the size of the index, and makes it less efficient. Set the fill factor appropriately, depending on the speed at which new data will be added.

Managing Statistics

What Are Statistics?

Statistics are used by the SQL Server query optimizer to create execution plans. SQL Server uses internal objects to hold information about the data that is held within a database; specifically, how values are distributed in columns or indexed views. Statistics that are out of date, or missing, can lead to poor query performance.

- Statistics are used by the query optimizer to estimate the cost of each execution plan
- Out-of-date or missing statistics lead to suboptimal query performance
- Statistics are automatically created and updated
- Use `sp_createstats` to update all single-column statistics in the database
- Regular maintenance on indexes does not affect statistics
- Updating statistics causes cached query plans to be recompiled

SQL Server calculates the statistics by knowing how many distinct values are held in a column.

Statistics are then used to estimate the number of rows that are likely to be returned at different stages of a query.

How Are Statistics Used in SQL Server?

The query optimizer is the part of SQL Server that plans how the database engine will execute a query. The query optimizer uses statistics to choose between different ways of executing a query; for example, whether to use an index or table scan. For statistics to be useful, they must be up to date and reflect the current state of the data.

Statistics can be updated automatically or manually, as required.

Automatically Update Statistics

You can determine how statistics are created and updated within your database by setting one of three options. These are specific to each database, and changed using the `ALTER DATABASE` statement:

- `AUTO_CREATE_STATISTICS`. The query optimizer creates statistics as required, to improve query performance. This is on by default and should normally stay on.

- AUTO_UPDATE_STATISTICS. The query optimizer updates out-of-date statistics as required. This is on by default and should normally stay on.
- AUTO_UPDATE_STATISTICS_ASYNC. This is only used when AUTO_UPDATE_STATISTICS is on. It allows the query optimizer to select an execution plan before statistics are updated. This is off by default.

Use ALTER DATABASE to set how statistics are updated in your database.

Alter Database Options

```
ALTER DATABASE AdventureWorks2016
SET AUTO_UPDATE_STATISTICS ON;
GO
```

There are two ways to create statistics manually: the CREATE STATISTICS command, or the sp_createstats stored procedure.

CREATE STATISTICS

You can create statistics on a single column, several columns, or filtered statistics. Using CREATE STATISTICS, you can specify:

- The name of the statistics object.
- The table or indexed view to which the statistics refer.
- The column or columns to be included in the statistics.
- The sample size on which the statistics should be based—this may be a scan of the whole table, a percentage of rows, or a count of rows.
- A filter for the statistics—filtered statistics are covered later in this lesson.
- Whether statistics should be created per-partition or for the whole table.
- Whether statistics should be excluded from automatic update.

 **Note:** Filtered statistics are created on a subset of rows in a table. Filtered statistics are created using the WHERE clause as part of the CREATE STATISTICS statement.

For more information about CREATE STATISTICS, see Microsoft Docs:

 **CREATE STATISTICS (Transact-SQL)**
<http://aka.ms/gdg0hw>

SP_CREATESTATS

If you want a quick way to create single-column statistics for all columns in a database that do not have a statistics object, you can use the sp_createstats stored procedure. This calls CREATE STATISTICS and is used for creating single-column statistics on all columns in a database not already covered by statistics. It accepts a limited selection of the options and parameters supported by CREATE STATISTICS.

For more information about sp_createstats, see Microsoft Docs:

 **Sp_createstats (Transact-SQL)**
<http://aka.ms/ak1mcd>

For more information about database statistics, see Microsoft Docs:



Statistics

<http://aka.ms/X36v84>

When to Update Statistics

You may need to update statistics manually, even when AUTO_UPDATE_STATISTICS is on, if statistics are not being updated often enough. The stale statistics detection method requires approximately 20 percent of table rows to change before statistics are updated. As a table grows, statistics will be updated less and less frequently, so you may want to update statistics manually on larger tables.

You could also update statistics manually as part of a work that makes large changes to the number of rows in a table; for example, bulk insert or truncating a table. This can improve the performance of queries because the query optimizer will have up-to-date statistics.

Index Maintenance

Index maintenance operations do not alter the distribution of data, and so do not require statistics to be updated. You do not need to update statistics after running ALTER INDEX REBUILD, DBCC REINDEX, DBCC INDEXDEFRAG, or ALTER INDEX REORGANIZE.

Statistics are automatically updated when you run ALTER INDEX REBUILD or DBCC DBREINDEX.



Best Practice: Do not update statistics more than necessary, because cached query execution plans will be marked for recompilation. Excessive updating of statistics may result in unnecessary CPU time being spent recompiling query execution plans.

Using DMOs to Improve Index Usage

Indexes play a crucial role in optimizing query performance. There are a number of dynamic management objects (DMOs) that provide helpful information about both current index usage, and missing indexes.

DMOs and Current Indexes

DMOs that provide information about current indexes:

- **sys.dm_db_index_usage_stats**—shows when and how indexes were last used. Use in combination with sys.objects. The information can be helpful to identify the following:
 - Indexes with high scan count and low seek count.
 - Unused indexes, which are not listed under this DMV.
 - Indexes with low seeks and high updates.
 - Frequently used indexes.
 - The last time the index was used.

- DMOs provide insight into existing and missing indexes
 - Existing indexes DMOs
 - sys.dm_db_index_usage_stats
 - sys.dm_db_physical_stats
 - sys.dm_db_index_operational_stats
 - Missing indexes DMOs
 - sys.dm_db_missing_index_details
 - sys.dm_db_missing_index_group_stats
 - sys.dm_db_missing_index_groups
 - sys.dm_db_missing_index_columns

- **sys.dm_db_physical_stats**—returns information about the index size and type, fragmentation, record count, and space used. Use to track the rate of fragmentation and to create an effective index maintenance strategy. Run during off-peak hours as this DMV can affect performance.
- **sys.dm_db_index_operational_stats**—returns I/O information, plus data about latches, locking, and waits.

For example, use **the sys.dm_db_index_physical_stats** DMV to get the level of fragmentation within an index.

sys.dm_db_index_physical_stats

```
Use AdventureWorks2016;
GO

SELECT * FROM sys.dm_db_index_physical_stats
    (DB_ID(N'AdventureWorks2016'), OBJECT_ID(N'Person.Address'), NULL, NULL ,
'DETAILED');
GO
```

DMOs and Missing Indexes

Missing indexes can have a huge impact on query performance. Some of the DMVs that provide information about missing indexes are:

- **sys.dm_db_missing_index_details**—identifies the table with missing index(es), and the columns required in an index. Does not work with spatial indexes. Test the index suggestion on a test server before deploying to production environment.
- **sys.dm_db_missing_index_group_stats**—shows details of how the index would be used
- **sys.dm_db_missing_index_groups**—a link DMV that links sys.dm_db_missing_index_details with sys.dm_db_missing_index_group_stats.
- **sys.dm_db_missing_index_columns**—a function that takes the index handle as an input, and returns the index columns required, and how they would be used.

For a list of index related DMOs, see Microsoft Docs:

 **Index Related Dynamic Management Views and Functions (Transact-SQL)**

<https://aka.ms/p5xu6z>

Consolidating Indexes

Although indexes are effective in improving the performance of queries, they come with a small overhead for inserts, deletes and updates. If you have two or more indexes that contain very similar fields, it might be worth consolidating the indexes into one larger index. This will mean that not all the columns will be used for each query; in turn, it will take a little more time to read. However, instead of updating several indexes, only one index will be updated when records are inserted, deleted, or updated.

- Indexes are effective in improving query performance
 - They have an update, insert, and delete overhead
- Consider merging similar indexes together
 - One larger index can be used by a number of queries
 - Takes more time to read
 - Only has to be updated once when data changes
- Measure the effect of your changes to ensure they have the intended results
- The order of columns in an index makes a difference

Even when indexes are used regularly, it might still be advantageous to consolidate them into one index, due to the reduced update overhead when changes occur. You should, however, measure the effect of changes you make to indexes, to ensure they have the results you intend, and that there are no unwanted side effects.

 **Note:** The order of columns in an index makes a difference. Although two indexes might have the same columns, they may produce very different performance results for different queries.

Using Query Hints

Query hints are signals to the query optimizer that a query should be executed in a certain way.

The query optimizer is sophisticated and, providing statistics are up to date, it will almost always find the best way of executing a query. Query hints should be the last thing, not the first, you try in tuning a query. Before attempting to add query hints, ensure the query is written properly, and that appropriate indexes are available.

- Query hints are signals to the query optimizer to execute a query in a certain way
- The query optimizer almost always figures out the best way to execute a query, providing:
 - Statistics are up to date
 - The query is properly constructed
- An error 8622 is raised if a query hint means the query optimizer cannot create a workable plan
- Query hints can be used to force a particular type of join, a recompile, or a maximum number of processors that can be used

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
Adding a query hint generates an error 8622. This means that the query optimizer cannot create a query plan.	

In the following example, the query optimizer would normally use a MERGE JOIN to join the two tables. In this code segment, a query hint is used to force a HASH JOIN. The cost of the join increases from 22 percent with a MERGE JOIN, to 67 percent with a HASH JOIN:

The following example uses a query hint to indicate that a HASH JOIN should be used:

Using a Query Hint

```
SELECT *
  FROM Production.Product AS P
    INNER JOIN Production.ProductDescription AS PD
      ON P.ProductID = PD.ProductDescriptionID
    WHERE SafetyStockLevel >= 500
OPTION (HASH JOIN);
GO
```

There are more than 20 query hints, including types of joins, the maximum number of processors to be used, and forcing a plan, to be recompiled. For a full list of query hints, see Microsoft Docs:

 **Query Hints (Transact-SQL)**

<http://aka.ms/t97w4b>



Best Practice: Use query hints judiciously. Unless there is a good reason to use a query hint, the query optimizer will find the best query plan.

Some query hints that usefully give the query optimizer additional information, and can be included, are:

- FAST *numberofrows*. This tells the query optimizer to retrieve the first n rows quickly, and then return the full result set. This may help provide a better user experience for some large result sets.
- OPTIMIZE FOR. Tells the query optimizer to use a specific variable when creating a plan. If you know that you will use one variable much more than any other, this may be useful. Other values are accepted when the query is executed.
- ROBUST PLAN. This tells the query optimizer to plan for the maximum row size, rather than for performance. This reduces errors when data has some very wide rows.

Check Your Knowledge

Question
When would you set FILL FACTOR for an index?
Select the correct answer.
When the index contains an IDENTITY column.
When a table is used frequently for data inserts, and is heavily used in queries.
When you want to force the query optimizer to join tables in a certain way.
When you are running short of disk space.
When you want to be sure that statistics are up to date.

Lesson 3

Execution Plans

Execution plans are generated before a SQL script is executed. They are generated by the query optimizer, and can help database developers and DBAs to understand why a query is being executed in a certain way.

Lesson Objectives

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

- What an execution plan is, and why it is useful.
- The difference between an actual and estimated execution plan.
- The different methods for capturing an execution plan.
- How to use system DMVs related to execution plans.
- What live statistics are, and how they are helpful.

What Is an Execution Plan?

How Execution Plans Help

Execution plans help you to understand why queries perform as they do. Database administrators (DBAs) are frequently faced with trying to understand:

- Why a particular query takes so long to execute.
- Why one query takes longer to run than a seemingly similar query.
- Why a newly created index doesn't make a difference to query performance.
- If a new index could help a query to run faster.

- Transact-SQL specifies what data is required; not how to retrieve it
- The query optimizer finds a good plan
 - Not always the very best plan
 - Good enough within a reasonable time
 - Not necessary for some DDL statements
 - Simple queries have a trivial plan
- It is a cost-based optimizer, and will compare a number of plans for each query
- The query optimizer uses statistics
- DDL statements do not normally have alternate plans

Execution plans help you to answer these questions by showing how the SQL Server Database Engine expects to execute a query, or how it has actually executed a query.

What Is the Query Optimizer?

Transact-SQL is a script language that tells SQL Server what data is required, rather than how to retrieve data. When you write a query, it is the job of the query optimizer to figure out the best way of handling it. The query optimizer is the part of SQL Server that predicts how the database engine will execute the query.

How the Query Optimizer Works

The query optimizer uses statistics to generate a cost-based plan for each query. To select the optimum plan, the query optimizer calculates a number of ways of executing the query. This is called the execution plan.

Trivial Plans

If a query is simple and executes quickly, the query optimizer does not plan a number of different ways to execute it—it just uses the most obvious plan. This is known as a trivial execution plan, because it is faster to execute the query than to compare a number of alternatives. Trivial queries normally retrieve data from a single table, and do not include calculations or aggregates.

Here is an example of a trivial query plan:

Simple Query with no Calculations or Aggregates

```
USE AdventureWorks2016
GO

SELECT *
FROM Production.Document
```

Adding a table join to this query would make the plan nontrivial. The query optimizer would then do cost-based calculations to select the best execution plan. You can identify trivial execution plans by running a system DMV (dynamic management view). Run the DMV before and after the query, noting the number of occurrences of trivial plans.

Count the number of trivial plans occurrences—before and after running your query.

sys.dm_exec_query_optimizer_info

```
SELECT * FROM sys.dm_exec_query_optimizer_info
```

Database Statistics

The query optimizer uses statistics to figure out how to execute a query. Statistics describe the data within the database, including its uniqueness. If statistics are out of date, the query optimizer will make incorrect calculations, and potentially choose a suboptimal query plan.

Cost-based Selection

The query optimizer uses a cost-based selection process to determine how to execute a query. The cost is calculated based on a number of factors, including CPU resources, memory, and I/O (input/output) operations, time to retrieve data from disk. It is not an absolute measure.

The query optimizer cannot try all possible execution plans. It has to balance the time taken to compare plans with the time taken to execute the query—and it has to cut off at a certain point. It aims to find a satisfactory plan within a reasonable period of time. Some data definition language (DDL) statements, such as CREATE, ALTER, or DROP, do not require alternative plans—they are executed straightaway.

Actual vs. Estimated Execution Plans

Selecting the Actual or Estimated Execution Plan

You can use SSMS to display either an actual or estimated execution plan for a query. Hover over the toolbar icons to display tool tips and find icons for **Display Estimated Execution Plan** and

Include Actual Execution Plan. Alternatively, right-click the query window and select either **Display Estimated Execution Plan** or **Include Actual Execution Plan** from the context-sensitive menu. You can also use the keyboard shortcuts **Ctrl-L** with the query highlighted for the estimated plan, or **Ctrl-M** to display the actual plan after the query has run. Whichever option you choose, SQL Server must generate an execution plan before the query can be executed.

- Display an execution plan by:
 - Clicking the icon on the toolbar or right-clicking the query window
 - Estimated execution plan (Ctrl-L)
 - Creates a plan but does not execute the query
 - Estimated number of rows based on statistics
 - Actual execution plan (Ctrl-M)
 - Mostly the same as the estimated plan
 - Includes actual number of rows returned
 - If different, statistics are out of date or missing
 - Query plans may be cached and reused

Estimated Execution Plan

The estimated execution plan is the query optimizer's intended plan. If the query is run straightaway, it will be no different from the actual plan, except that it has not yet run. Estimated execution plans do not include the actual number of rows at each stage. The estimated execution plan is useful for:

- Designing or debugging queries that take a long time to run.
- Designing a query that modifies data; for example, a query that includes an UPDATE statement. The estimated execution plan will display the plan without changing the data.
- SQL Server shows estimates for the number of rows returned.

Actual Execution Plan

The actual execution plan is displayed after a query has been executed. This displays actual data, including the number of rows accessed for a particular operator. The plan is normally the same as the estimated plan, but with additional information. If there are missing or out-of-date statistics, the plans may be different.

Hover over each part of the actual execution plan to display more information about how the query was executed.



Note: Comparing estimated and actual row counts can help you to identify out-of-date table statistics. When statistics are up to date, estimated and actual counts will be the same.

Query Plan Cache

Execution plans are stored in the plan cache so they can be reused. Execution plan reuse saves time and allows queries to run faster, because the query optimizer does not have to consider alternative plans.

For more information about optimizing queries using the plan cache, see MSDN:



Plan Cache Internals

<http://aka.ms/Vsiip6>

Common Execution Plan Elements

Execution plans include logical or physical operators to build the query plan. More than 100 different operators may appear in an execution plan. Operators can have an output stream of data rows, in addition to zero, one, or two input data streams. Execution plans are read from right to left.

Many operators are represented with an icon in the graphical execution plan display. Operators can be categorized according to the function. Commonly-used categories include:

- Data retrieval operators
- Join operators
- Parallelized query plans

- Data retrieval operators
 - Scan: reads records sequentially
 - Seek: finds the appropriate record in an index
 - Join operators
 - Nested Loop: the second input is searched once for each value in the first input; the second input is inexpensive to search
 - Merge Join: two sorted inputs are interleaved
 - Hash Match: a hash table is built from the first input; this is compared against hash values from the second input—typically, large, unsorted inputs
 - Parallel query plans have at least one instance of the Gather Streams operator

Data Retrieval Operators

Queries that retrieve data from database tables will have a query plan that includes a scan or seek operator.

- **Scan:** a scan reads records sequentially and retrieves the required records. A scan is used when a large proportion of records are retrieved from a table, or if there is no suitable index.
- **Seek:** a seek finds specific records by looking them up in an index. This is normally faster than a scan, because specific records can be quickly located and retrieved using the index.

A scan may be used on a clustered index, a nonclustered index, or a heap. A seek may be used on a clustered or nonclustered index. Both scan and seek operators may output some or all of the rows they read, depending on what filters are required for the query.

A query plan can always use a scan, but a seek is used only when there is a suitable index. Indexes that contain all the columns required by a query are called covering indexes.

 **Best Practice:** The query execution plan is a guide to help you to understand how queries are being executed. Do not, however, try to manipulate how the query optimizer handles a query. When table statistics are accurate, and appropriate indexes are available, the query optimizer will almost always find the fastest way of executing the query.

Join Operators

JOIN clauses are used in queries that retrieve records from more than one table. The query execution plan includes join operators that combine data that is returned by scans or seeks. The data is transformed into a single data stream by the join operators.

The query optimizer uses one of three join operators, each of which takes two input data streams and produces one output data stream:

- Nested loop
- Merge join
- Hash match

Nested Loop

A nested loop join performs a search from the second input data stream for each row in the first input data stream. Where the first input data stream has 1,000 rows, the second input will be searched once for each row—so it performs 1,000 searches.

In a graphical query plan, the upper input is the first input and the lower input is the second input. In an XML or text query plan, the second input will appear as a child of the first input. Nested loop joins are used when the second input is inexpensive to search, either because it is small or has a covering index.

Merge Join

A merge join combines two sorted inputs by interleaving them. The sequence of the input streams has no impact on the cost of the join. Merge joins are optimal when the input data streams are already sorted, and are of similar volumes.

Hash Match

A hash match calculates a hash value for each input data stream, and the hash values are compared. The operation details vary according to the source query, but typically a complete hash table is calculated for the first input, then the hash table is searched for individual values from the second input. Hash matches are optimal for large, unsorted input data streams, and for aggregate calculations.

Parallel Query Plans

When multiple processors are available, the query optimizer might attempt to speed up queries by running tasks in parallel on more than one CPU. This is known as parallelism and normally involves large numbers of rows.

The query execution plan does not actually show the individual threads participating in a parallel query plan; however, it does show a logical sequence of operators, and the operators that use parallelism are flagged.

In a graphical plan, parallelized operators have a small orange circle containing two arrows overlaid on the bottom right-hand corner of the icon. In XML query plans, parallelized operators have the “parallel” attribute set to “true”. In text query plans generated by SHOWPLAN_ALL or STATISTICS PROFILE, the result set contains a parallel column with a value of 1 for parallelized operators.

Parallel query plans will also contain at least one instance of the Gather Streams operator, which combines the results of parallelized operators.

For more information about query plan operators, see Microsoft Docs:

 **Showplan Logical and Physical Operators Reference**

<http://aka.ms/ifow8r>

Methods for Capturing Plans

The way you capture a query execution plan varies according to the format in which the plan is displayed.

 **Note:** Capturing a query execution plan requires the SHOWPLAN permission, as well as permission to the objects referenced in the query. You cannot generate an execution plan for a query that you do not have permission to execute.

- Graphical plan
 - Right-click and Save Execution Plan As
 - Saved in XML format with a .sqlplan extension
 - .sqlplan is associated with SSMS
- Use SET SHOWPLAN_XML ON to display an estimated execution plan as an XML document
- USE SET STATISTICS XML ON to display the actual execution plan as an XML document

Graphical Execution Plan

To save a graphical execution plan, right-click the graphical execution plan displayed in the results window, and click **Save Execution Plan As**. The Save As dialog appears, allowing you to name the plan and save it to an appropriate folder. The plan will be saved in XML format, with a .sqlplan extension. By default, this extension is associated with SSMS and will open in SSMS.

To view a graphical plan in XML format, right-click the plan and click **Show Execution Plan XML**. The execution plan is already in XML format, but is displayed graphically by default.

XML Execution Plan

Use the SET SHOWPLAN_XML option to display an estimated execution plan in XML format.

Generate an estimated execution plan in XML format by using the SHOWPLAN_XML option.

SET SHOWPLAN_XML ON

```
USE AdventureWorks2016;
GO

-- Display an estimated execution plan in XML format
SET SHOWPLAN_XML ON;
GO

SELECT *
    FROM HumanResources.Employee
    WHERE gender = 'F';
GO
```

Use the SET STATISTICS XML ON to display the actual execution plan in XML format.

Display the actual execution plan in XML format by using the SET STATISTICS XML ON option.

SET STATISTICS XML ON

```
USE AdventureWorks2016;
GO

-- Display the actual execution plan in XML format
SET STATISTICS XML ON;
GO

SELECT *
    FROM HumanResources.Employee
```



Note: The toolbar icon **Include Actual Execution Plan** and the **Ctrl-M** keyboard shortcut will toggle, showing the actual execution plan on and off. Ensure this is off before running the SET statements.

Execution Plan Related DMVs

In addition to query execution plans, SQL Server also stores performance statistics for cached plans, such as execution time, I/O activity, and CPU utilization. These statistics are accessible through system DMVs to help you troubleshoot performance problems, or identify areas that require optimization.

- **sys.dm_exec_query_stats** returns performance information for all the cached plans.
- **sys.dm_exec_procedure_stats** returns the same information as sys.dm_exec_query_stats, but for stored procedures only.

- Use system DMVs to return information captured for cached plans:
 - Sys.dm_exec_query_stats
 - Sys.dm_exec_procedure_stats
- Use these DMVs to troubleshoot, and identify candidates for optimization and performance tuning

Use this query to find the top 10 cached plans with the highest average run time per execution.

Use DMVs To Find the Longest Running Queries

```
SELECT TOP(10) OBJECT_NAME(st.objectid, st.dbid) AS obj_name, qs.creation_time,
       qs.last_execution_time, SUBSTRING (st.[text], (qs.statement_start_offset/2)+1,
       (( CASE statement_end_offset WHEN -1 THEN DATALENGTH(st.[text])
             ELSE qs.statement_end_offset
             END - qs.statement_start_offset)/2)+1) AS sub_statement_text, [text],
       query_plan, total_worker_time, qs.execution_count,qs.total_elapsed_time /
       qs.execution_count AS avg_duration

       FROM sys.dm_exec_query_stats AS qs
       CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
       CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) AS qp

       ORDER BY avg_duration DESC;
```

Use the same query with a small amendment to find the 10 cached plans that use the highest average CPU per execution.

Amendment to Find Top Average CPU Consumption

```
SELECT TOP(10) OBJECT_NAME(st.objectid, st.dbid) AS obj_name, qs.creation_time,
       qs.last_execution_time, SUBSTRING (st.[text], (qs.statement_start_offset/2)+1,
       (( CASE statement_end_offset WHEN -1 THEN DATALENGTH(st.[text])
             ELSE qs.statement_end_offset
             END - qs.statement_start_offset)/2)+1) AS sub_statement_text,
       [text], query_plan, total_worker_time, qs.execution_count, qs.total_worker_time /
       qs.execution_count AS avg_cpu_time,
       qs.total_elapsed_time / qs.execution_count AS avg_elapsed_time

       FROM sys.dm_exec_query_stats AS qs
       CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
       CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) AS qp

       ORDER BY avg_cpu_time DESC;
```



Note: The unit for time columns in sys.dm_exec_query_stats (for example total_worker_time) is microseconds (millionths of a second). However, the values are only accurate to milliseconds (thousandths of a second).

Amend the example to find the 10 most expensive queries by the average logical reads per execution.

Top Ten Most Expensive Cached Plans by Average Logical Read

```
SELECT TOP(10) OBJECT_NAME(st.objectid, st.dbid) AS obj_name, qs.creation_time,
qs.last_execution_time,
SUBSTRING (st.[text], (qs.statement_start_offset/2)+1,
(( CASE statement_end_offset WHEN -1 THEN DATALENGTH(st.[text])
ELSE qs.statement_end_offset
END - qs.statement_start_offset)/2)+1) AS sub_statement_text,
[text], query_plan, total_worker_time, qs.execution_count,
qs.total_logical_reads / qs.execution_count AS avg_logical_reads,
qs.total_elapsed_time / qs.execution_count AS avg_duration

FROM sys.dm_exec_query_stats AS qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) AS st
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) AS qp

ORDER BY avg_logical_reads DESC;
```

You can adapt these queries to return the most expensive queries by any of the measures in sys.dm_exec_query_stats, or to limit the results to stored procedure cached plans by using sys.dm_exec_procedure_stats in place of sys.dm_exec_query_stats.

For more details, see *sys.dm_exec_query_stats (Transact-SQL)* in Microsoft Docs:



[sys.dm_exec_query_stats \(Transact-SQL\)](http://aka.ms/n3fhua)

<http://aka.ms/n3fhua>

Remember, **sys.dm_exec_query_stats** only returns information about cached plans. Plans that are not in the cache, either because they have been recompiled, or because they have not been executed again since the plan cache was last flushed, will not appear.

Live Query Statistics

Live Query Statistics

Use Live Query Statistics to obtain a real-time view of how a query is executed. The execution plan is just a plan—the database engine may execute the query differently—but, with Live Query Statistics, you can see statistics as the query is executing.

Live Query Statistics can be used with databases developed using SQL Server 2014 or later. Live Query Statistics is a feature of SQL Server Management Studio, and you should download the most recent version of SSMS independently of the database engine. See Microsoft Docs:

- Live Query Statistics gives you a real-time view of how a query is being executed
- Works with SQL Server 2014 and later database engine
- Download the latest version of SSMS
- You cannot use Live Query Statistics with natively compiled stored procedures

 **Download SQL Server Management Studio (SSMS)**

<http://aka.ms/o4vgkz>

Displaying Live Query Statistics

There are two ways of including Live Query Statistics when you run your query:

1. Click the **Include Live Query Statistics** icon on the toolbar. The icon is highlighted to show that it is selected. Execute the query—the **Live Query Statistics** tab is displayed.
2. Right-click the query window and select **Include Live Query Statistics** from the context-sensitive menu. Execute the query—the **Live Query Statistics tab** is displayed.

By including Live Query Statistics using either method, you are enabling statistics for the current session. If you want to view Live Query Statistics from other sessions, including Activity Monitor, you must execute one of the following statements:

- SET STATISTICS XML ON
- SET STATISTICS PROFILE ON

Alternatively, you can use **query_post_execution_showplan** to enable the server setting. See Microsoft Docs:

 **Monitor System Activity Using Extended Events**

<http://aka.ms/N8n81l>

What Does Live Query Statistics Show?

Live Query Statistics differs from viewing an execution plan, in that you can see query progress. Execution plans show you either the expected query plan, or how the plan was actually executed—but neither provides an in-progress view. Live Query Statistics shows you the numbers of rows produced by each operator, and the elapsed time for each operation.

For more information about Live Query Statistics, see Microsoft Docs:

 **Live Query Statistics**

<http://aka.ms/Pvznef>

 **Note:** Live Query Statistics is a troubleshooting tool, and should be used for optimizing queries. It adds an overhead to the query, and can affect performance.

Question: Why would you use the execution plan view?

Lesson 4

The Database Engine Tuning Advisor

The Database Engine Tuning Advisor analyzes a workload and makes recommendations to improve its performance. You can use either the graphical view, or the command-line tool to analyze a trace file.

Lesson Objectives

After completing this lesson, you will be able to:

- Understand what the Database Engine Tuning Advisor is, and when to use it.
- Explain how to use the Database Engine Tuning Advisor.

Introduction to the Database Engine Tuning Advisor

The Database Engine Tuning Advisor is a tool to help you improve the performance of your queries. It is a stand-alone tool that must be started independently of SSMS.

The Database Engine Tuning Advisor may recommend:

- New indexes or indexed views.
- Statistics that need to be updated.
- Aligned or nonaligned partitions.
- Better use of existing indexes.

- The Database Engine Tuning Advisor helps you improve query performance
- It is a stand-alone tool
- It makes recommendations, such as:
 - New indexes or indexed views
 - Statistics that need to be updated
 - Aligned or nonaligned partitions
 - Make better use of existing indexes
- The Database Engine Tuning Advisor uses a saved workload

Before you can use the Database Engine Tuning Advisor, you must capture a typical workload.



Note: The Database Engine Tuning Advisor is not widely used, although it might be useful in some circumstances. New tools such as Query Store, discussed later in this module, may be easier to use for most situations.

Using the Database Engine Tuning Advisor

Using the Database Engine Tuning Advisor for the First Time

The first time Database Engine Tuning Advisor is used, it must be started by a member of the sysadmin role, because system tables are created in the **msdb** database.

- First-time use must be by a member of the sysadmin role
- Tables are created in the msdb database
- Accepts different workload formats:
 - Plan cache
 - SQL Profiler trace
 - Transact-SQL script
 - XML file

Workload Formats

The Database Engine Tuning Advisor can accept any of the following workloads:

- Plan Cache
- SQL Profiler tracer file or table

- Transact-SQL script
- XML file

For more information about using the Database Engine Tuning Advisor, see Microsoft Docs:

 **Start and Use the Database Engine Tuning Advisor**

<http://aka.ms/Mdpzxr>

Check Your Knowledge

Question	
The Database Engine Tuning Advisor is best used in which situations?	
Select the correct answer.	
	When you need to add a large number of records to a table.
	When you want to ensure indexes are rebuilt regularly.
	When you want to identify missing indexes.
	When you need to create an XML file.
	Every time you run a Transact-SQL script.

Lesson 5

Query Store

SQL Server includes Query Store, a feature that makes it easier to find and fix problem queries. This lesson introduces the Query Store, how to enable it, and how to use it.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the benefits of using Query Store.
- Enable and configure Query Store for a database.
- Use the different Query Store views to monitor query performance.

What Is the Query Store?

Query Store helps you to improve the performance of troublesome queries. Query Store is available in all SQL Server editions, and Azure SQL Database.

It provides better insight into the way queries are executed, in addition to capturing a history of every query that is run on the database. Query Store also keeps every version of the execution plan used by each query, plus execution statistics.

- Query Store is in all editions of SQL Server
 - SQL Server 2016 and later
 - Already in Azure SQL Database
- Previously, troublesome queries were time-consuming to fix
- Query Store is a permanent record of query execution plans
- Data is stored in the user database
- Query plan data survives reboots, failovers, and database upgrades

What Problems Does Query Store Solve?

Query performance problems often occur when the query plan changes—perhaps because statistics have been updated, the database version has been upgraded, or the query cache has been cleared. Slow running queries can cause big problems with applications or websites, potentially making them unusable. Even temporary performance problems can create a lot of work in trying to understand what has caused the issue.

The most recent execution plan is stored in the plan cache but, prior to Query Store, it could be time-consuming and difficult to work out what was causing a problem with query performance. Query Store has been designed to make it easier to understand and fix problem queries.

What Does Query Store Do?

Query Store is a repository for all your queries, execution plans, properties, and metrics. You can see how long queries took to run, in addition to which execution plan was used by the query. You can also force the query optimizer to use a particular execution plan. In short, Query Store keeps data that makes it easier to solve query problems.

How Does Query Store Work?

The information is stored in each user database, with settings to control when old queries should be deleted. Query Store writes query information to disk, so a permanent record is kept after failovers, server restarts, and database upgrades. This makes it simpler to identify and fix the most problematic queries, and analyze query performance.

For more information about Query Store, see Microsoft Docs:

Monitoring Performance By Using the Query Store

<http://aka.ms/M1iwvm>

Enabling Query Store

Before you can use Query Store, you must enable it for your database. This can be done either by using a database properties dialog box, or by using Transact-SQL.

Amend Database Properties

To enable Query Store using the GUI, start SSMS and select the relevant database in Object Explorer. Right-click the database name and select **Properties** from the context-sensitive menu.

Amend the first setting under **General** to **Read Write**. By default, this is set to off.

- Query Store must be enabled on each database
- Use Object Explorer, then Database Properties
 - Set Operation Mode to Read Write
- Transact-SQL ALTER DATABASE
 - SET QUERY_STORE = ON
- Query Store options include:
 - Max disk space
 - Max plans per query
 - When to flush in-memory data to disk
- Use sys.database_query_store_options to view the settings

Transact-SQL ALTER DATABASE

This is done using the ALTER DATABASE statement.

Use ALTER DATABASE to enable Query Store for your database.

SET QUERY_STORE = ON

```
ALTER DATABASE AdventureWorks2016
SET QUERY_STORE = ON;
GO
```

Query Store works well with the default settings, but you can also configure a number of other settings. If you have a high workload, configure Query Store to behave when it gets close to disk space capacity.

Query Store Options

Set or amend these settings using ALTER DATABASE.

Setting	Options	Default	Comments
OPERATION_MODE	OFF, READ_WRITE or READ_ONLY	OFF	When Query Store is running, this should be set to Read Write. It might automatically get set to Read Only if it runs out of disk space.
CLEANUP_POLICY	STALE_QUERY _THRESHOLD_DAYS	367	Determines how long a query with no policy is kept. Set to 0 to disable.
DATA_FLUSH_INTERVAL_SECONDS	Number of seconds	900 (15 mins)	The interval at which SQL Server writes in memory data to disk.
MAX_STORAGE_SIZE_MB	Number of MB	100 MB	The maximum disk space in MB reserved for Query Store.

Setting	Options	Default	Comments
INTERVAL_LENGTH_MINUTES	Number of minutes	60 mins	Determines how statistics are aggregated.
SIZE_BASED_CLEANUP_MODE	AUTO or OFF	OFF	Auto means that the oldest queries are deleted once 90 percent of disk space is filled. Queries are no longer deleted when there is 80 percent of disk space. OFF is the default and no more queries will be captured when the available disk space is full. Operation mode switches to Read Only.
QUERY_CAPTURE_MODE	ALL, AUTO or NONE	ALL	ALL means that all queries will be captured. AUTO captures queries with high execution count and resource consumption. NONE does not capture queries.
MAX_PLANS_PER_QUERY	Number	200	Determines the number of plans that can be kept per query.
QUERY_STORE	ON, OFF or CLEAR		Determines whether or not Query Store is enabled for a database. Clear removes the contents of Query Store.

You can view the settings either by using database properties in SSMS Object Explorer, or by using Transact-SQL.

View the Query Store settings using `sys.database_query_store_options`.

`sys.database_query_store_options`

```
SELECT *
FROM sys.database_query_store_options;
```



Note: Query Store is only used with user databases, it cannot be enabled for **master**, **msdb**, or **tempdb**.

Using Query Store

After Query Store has been enabled, a Query Store folder for the database is created in Object Explorer. Query Store then starts to store information about queries that have been executed. Data collected by the Query Store can be accessed, either by using SSMS, or by using Transact-SQL queries.

Query Store in SSMS

The Query Store folder in SSMS includes four views that show different information:

1. Regressed Queries.
2. Overall Resource Consumption.
3. Top Resource Consuming Queries.
4. Tracked Queries.

Regressed Queries

This view shows queries whose performance has degraded over a period of time. You use a drop-down box to select performance measured by CPU time, duration, logical round count, logical write count, memory consumption, or physical reads. You can also see the execution plan for each query.

Overall Resource Consumption

This view displays resource consumption over time. Histograms can show consumption by CPU time, duration, logical read count, logical write count, memory consumption, or physical reads.

Top Resource Consuming Queries

This view shows the top 25 most expensive queries over time by CPU time, duration, logical read count, logical write count, memory consumption, or physical reads. You can also see the execution plan for each query.

Tracked Queries

This view shows historical data for a single query.

Query Store using Transact-SQL

In addition to the graphical view, system catalog views are available to expose the data collected by the Query Store. These catalog views are similar to the query plan cache DMVs discussed earlier in this module. Some of the Query Store catalog views are:

- sys.query_store_runtime_stats. Similar to sys.dm_exec_query_stats, this catalog view exposes performance information captured by Query Store.
- sys.query_store_plan. Similar to sys.dm_exec_query_plan, this catalog view exposes query plans data captured by Query Store.
- sys.query_store_query_text. Similar to sys.dm_exec_query_text, this catalog view exposes the query text ID and the query text relating to queries captured by the Query Store.

For more information on Query Store catalog views, see MSDN:

-  [Query Store Catalog Views \(Transact-SQL\)](http://aka.ms/u60m6c)
<http://aka.ms/u60m6c>

- Use Query Store through SSMS Object Explorer:
 - Query Store node under the relevant database
- Four views:
 - Regressed queries
 - Overall resource consumption
 - Top resource consuming queries
 - Tracked queries
- Transact-SQL system catalog views:
 - sys.query_store_runtime_stats
 - sys.query_store_plan
 - sys.query_store_query_text

Improving Query Performance

After you have identified the queries that are taking the longest time to run, you can use the tools in the Query Store to discover why this happens. You can use the Top Resource Consuming Queries view to see the query plans used by the query over time, and to view the associated execution plan. The execution plan shows you where there are issues in your query: perhaps where you have joined to a column without an index, or used a function that is causing the query to slow down. Using this view, you can force a query to use the most efficient plan.

- Use the Top Resource Consuming Queries view to find long-running queries
- View the execution plan for the query to find the cause of low performance
- Force a query to use a plan, or:
 - Rewrite the query until the duration decreases to an acceptable time
 - Query the data captured by the Query Store by using the `sys.query_store_plan` view

The following code sample demonstrates how you can directly query the data captured by the Query Store:

Querying the Query Store System Views

```
SELECT plan_id, query_id, [compatibility_level], is_forced_plan,
       count_compiles, last_compile_start_time, last_execution_time
  FROM sys.query_store_plan
```

Demonstration: Using Query Store with Azure SQL Database

Demonstration Steps

1. Ensure the **MT17B-WS2016-NAT**, **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running, and log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Open SSMS and connect to the server you created earlier—for example, **20762ce20160405.database.windows.net**, using SQL Server Authentication.
3. In the **Login** box, type **Student**, in the **Password** box, type **Pa55w.rd**, and then click **Connect**.
4. In Object Explorer, expand **Databases** right-click **AdventureWorksLT**, and then click **Properties**.
5. In the **Database Properties - AdventureWorksLT** dialog box, click the **Query Store** page, and in the **General** section, ensure the **Operation Mode (Requested)** is set to **Read Write**. Point out the Query Store settings to students. When you are finished, click **OK**.
6. In Object Explorer, expand the **AdventureWorksLT** node to see that a folder called **Query Store** has been created.
7. Expand the **Query Store** node to show the four views.
8. On the **File** menu, point to **Open**, and then click **File**.
9. In the **Open File** dialog box, navigate to **D:\Demofiles\Mod06**, and open **QueryStore_Demo.sql**.
10. Select the code under the comment **Create a covering index on the TempProduct table**, and then click **Execute**. First, the query creates a covering index on the **TempProduct** table, and then uses three columns from this table—point out that the text columns have been included as nonkey columns.

11. Select the code under the comment **Clear the Query Store**, and then click **Execute**.
12. Select the code under the comment **Work load 1**, and then click **Execute**.
13. Repeat the previous step another five times, waiting a few seconds between each execution.
14. Select the code under the comment **Work load 2**, and then click **Execute**.
15. Repeat the previous step another three times, waiting a few seconds between each execution.
16. Select the code under the comment **Regress work load 1**, and then click **Execute**.
17. Select the code under the comment **Work load 1**, and then click **Execute**.
18. Repeat the previous step another five times, waiting a few seconds between each execution.
19. In Object Explorer, open the **Top Resource Consuming Queries** window, and see the difference between the two execution plans for Workload 1.
20. Demonstrate how you can change the metric using the drop-down box. Note the force plan button.
21. On the **File** menu, point to **Open**, and then click **File**.
22. In the **Open File** dialog box, navigate to **D:\Demofiles\Mod06**, select **QueryStore_Demo_CatalogViews.sql**, and then click **Open**.
23. Select the code under the comment **Create a covering index on the TempProduct table**, and then click **Execute**.
24. Select the code under the comment **Clear the Query Store**, and then click **Execute**.
25. Select the code under the comment **Work load 1**, and then click **Execute**.
26. Repeat the previous step.
27. Select the code under the comment **Work load 2**, and then click **Execute**.
28. Repeat the previous step another two times, waiting a few seconds between each execution.
29. Select the code under the comment **Regress work load 1**, and then click **Execute**.
30. Select the code under the comment **Work load 1**, and then click **Execute**.
31. Select the code under the comment **Examine sys.query_store_query_text and sys.query_context_settings**, and then click **Execute**.
32. Select the code under the comment **Examine sys.query_store_query**, and then click **Execute**.
33. Select the code under the comment **Examine sys.query_store_plan**, and then click **Execute**.
34. Select the code under the comment **Examine sys.query_store_runtime_stats_interval**, and then click **Execute**.
35. Select the code under the comment **Examine runtime statistics**, and then click **Execute**.
36. Close SSMS without saving any changes.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? Due to limitations of using the cloud, Azure SQL Database does not contain all the Query Store functionality that is available in the SQL Server on-premises product.	

Lab: Optimizing Indexes

Scenario

You have been hired by the IT Director of the Adventure Works Bicycle Company to work with their DBA to improve the use of indexes in the database. You want to show the DBA how to use Query Store to improve query performance and identify missing indexes. You will also highlight the importance of having a clustered index on each table.

Objectives

In this lab, you will practice:

- Using Query Store to monitor queries and identify missing indexes.
- Compare a heap against a table with a clustered index.

Estimated Time: 45 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Using Query Store

Scenario

You are the DBA for the Adventure Works Bicycle Company. You have been working with a consultant to implement the features in Query Store, and now want to simulate a typical query load.

The main tasks for this exercise are as follows:

1. Use Query Store to Monitor Query Performance

► Task 1: Use Query Store to Monitor Query Performance

1. Log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab06\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In SSMS, connect to the **MIA-SQL** database engine instance using Windows authentication.
4. Open **QueryStore_Lab1.sql**.
5. Make **AdventureWorks2016** the current database.
6. Execute the code to create an indexed view.
7. Select the code under the comment **Clear the Query Store**, and then click **Execute**.
8. Select the code under the comment **Run a select query six times**, and then click **Execute**.
9. Repeat five times, waiting a few seconds each time.
10. Select the code to **Update the statistics with fake figures**, and then click **Execute**.
11. Repeat Step 10 twice.
12. In **Query Store**, double-click **Top Resource Consuming Queries**.
13. Examine the different views of the Top 25 Resources report and force the original query plan to be used.
14. Switch to the **QueryStore_Lab1.sql** tab and repeat Step 10 three times.

15. Switch to the **Top Resource Consuming Queries** tab to identify which query plans used a clustered index seek and which ones used a clustered index scan.
16. Keep SSMS open for the next lab exercise.

Results: After completing this lab exercise you will have used Query Store to monitor query performance, and used it to force a particular execution plan to be used.

Exercise 2: Heaps and Clustered Indexes

Scenario

You are the DBA for the Adventure Works Bicycle Company. You have had complaints that a number of queries have been running slowly. When you take a closer look, you realize that a number of tables have been created without a clustered index. Before adding a clustered index, you decide to run some tests to find out what difference a clustered index makes to query performance.

The main tasks for this exercise are as follows:

1. Compare a Heap with a Clustered Index
 - ▶ **Task 1: Compare a Heap with a Clustered Index**
 1. Open **ClusterVsHeap_lab.sql**, and run each part of the script in turn.
 2. Make **AdventureWorks2016** the current database.
 3. Run the script to create a table as a heap.
 4. Run the script to create a table with a clustered index.
 5. Run the script to SET STATISTICS ON. Run each set of select statements on both the heap, and the clustered index.
 6. Open **HeapVsClustered_Timings.docx**, and use the document to note the CPU times for each.
 7. Run the script to select from each table.
 8. Run the script to select from each table with the ORDER BY clause.
 9. Run the script to select from each table with the WHERE clause.
 10. Run the script to select from each table with both the WHERE clause and ORDER BY clause.
 11. Run the script to insert data into each table.
 12. Compare your results with the timings in the Solution folder.
 13. If you have time, run the select statements again and **Include Live Query Statistics**.
 14. Close SQL Server Management Studio, without saving any changes.
 15. Close Wordpad.

Results: After completing this lab exercise, you will:

Understand the effect of adding a clustered index to a table.

Understand the performance difference between a clustered index and a heap.

Question: Which Query Store features will be most beneficial to your SQL Server environment?

Question: In which situation might a heap be a better choice than a table with a clustered index?

Question: Why is it sometimes quicker to retrieve records from a heap than a clustered index with a simple SELECT statement?

Module Review and Takeaways

Indexes are one of the most important tools in improving the efficiency of queries, and the performance of your database. However, we have seen that each index has a cost in terms of additional overhead when inserting records into a table—in addition to maintenance as leaf-level pages are filled. Use FILL FACTOR and PAD INDEX appropriately, depending on the structure of your table, and how much data will be added.

Use the Query Store to understand how the most resource intensive queries are performing, and to take corrective action before they become a problem. Because it stores historical query plans, you can compare them over time to see when and why a plan has changed. After enabling the Query Store on your databases, it automatically runs in the background, collecting run-time statistics and query plans; it also categorizes queries, so it is easy to find those using the most resources, or the longest-running operations. Query Store separates data into time windows, so you can uncover database usage patterns over time.



Best Practice: Understand how queries are being executed using the estimated and actual execution plans, in addition to using Query Store. When you need to optimize a query, you will then be well prepared and have a good understanding of how SQL Server executes your Transact-SQL script.

Module 7

Columnstore Indexes

Contents:

Module Overview	7-1
Lesson 1: Introduction to Columnstore Indexes	7-2
Lesson 2: Creating Columnstore Indexes	7-7
Lesson 3: Working with Columnstore Indexes	7-12
Lab: Using Columnstore Indexes	7-17
Module Review and Takeaways	7-21

Module Overview

Introduced in Microsoft® SQL Server® 2012, columnstore indexes are used in large data warehouse solutions by many organizations. This module highlights the benefits of using these indexes on large datasets, the improvements made to columnstore indexes in the latest versions of SQL Server, and the considerations needed to use columnstore indexes effectively in your solutions.

Objectives

After completing this module, you will be able to:

- Describe columnstore indexes and identify suitable scenarios for their use.
- Create clustered and nonclustered columnstore indexes.
- Describe considerations for using columnstore indexes.

Lesson 1

Introduction to Columnstore Indexes

This lesson provides an overview of the types of columnstore indexes available in SQL Server; the advantages they have over their similar row based indexes; and under what circumstances you should consider using them. By the end of this lesson, you will see the potential cost savings to your business of using clustered columnstore index, purely in terms of the gigabytes of disk storage.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the differences between rowstore and columnstore indexes.
- Describe the properties of nonclustered columnstore indexes.
- Describe the properties of a clustered columnstore index and how it differs from a nonclustered columnstore index.

What are Columnstore Indexes?

Columnstore indexes reference data in a columnar fashion, and use compression to reduce the disk I/O when responding to queries.

Traditional rowstore tables are stored on disk in pages; each page contains a number of rows and includes all the associated columns with each row. Columnstore indexes also store data in pages, but they store all the column values in a page—so the page consists of the same column of data from multiple rows.

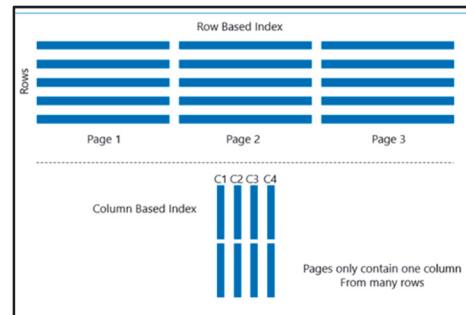
Consider a data warehouse containing fact tables that are used to calculate aggregated data across multiple dimensions. These fact tables might consist of many rows, perhaps numbering tens of millions.

Totaling Sales Orders by Product

Using a code example:

Totaling Sales Orders by Product

```
SELECT ProductID,
       SUM(LineTotal) AS ProductTotalSales
  FROM Sales.OrderDetail
 GROUP BY ProductID
 ORDER BY ProductID
```



Thinking about the previous example using a row based index, it will need to load into memory all the rows and columns in all the pages, for all the products. With a column based index, the query only needs to load the pages associated with the two referenced columns, ProductID and LineTotal. This makes columnstore indexes a good choice for large data sets.

Using a columnstore index can improve the performance for a typical data warehouse query by up to 10 times. There are two key characteristics of columnstore indexes that impact this gain.

- **Storage.** Columnstore indexes store data in a compressed columnar data format instead of by row. This makes it possible to achieve compression ratios of seven times greater than a standard rowstore table.
- **Batch mode execution.** Columnstore indexes process data in batches (of 1,000-row blocks) instead of row by row. Depending on filtering and other factors, a query might also benefit from “segment elimination,” which involves bypassing million-row chunks (segments) of data and further reducing I/O.

Columnstore indexes perform well because:

- Columns often store matching data—for example, a set of States enabling the database engine to compress the data better. This compression can reduce or eliminate any I/O bottlenecks in your system, while also reducing the memory footprint as a whole.
- High compression rates improve overall query performance because the results have a smaller in-memory footprint.
- Instead of processing individual rows, batch execution also improves query performance. This can typically be a performance improvement of around two to four times because processing is undertaken on multiple rows simultaneously.
- Aggregation queries often select only a few columns from a table, which also reduces the total I/O required from the physical media.

Nonclustered and clustered indexes are supported in Azure® SQL Database Premium Edition. For a full list of the columnstore features available in different versions of SQL Server, see Microsoft Docs:

Columnstore Indexes Versioned Feature Summary

<http://aka.ms/gj0fkY>

There are two types of columnstore indexes—nonclustered and clustered columnstore indexes—that both function in the same way. The difference is that a nonclustered index will normally be a secondary index created on top of a rowstore table; a clustered columnstore index will be the primary storage for a table.

Nonclustered Columnstore Indexes

Nonclustered columnstore indexes contain a copy of part or all of the columns in an underlying table. Because this kind of index is a copy of the data, one of the disadvantages is that it will take up more space than if you were using a rowstore table alone.

A nonclustered columnstore index has the following characteristics:

- It can include some or all of the columns in the table.
- It can be combined with other rowstore indexes on the same table.

- Nonclustered Columnstore Indexes
 - Contains some or all columns
 - Used in combination with rowstore tables
 - Updatable
 - Can be filtered
 - Uses more space than just a rowstore

- It is a full or partial copy of the data and takes more disk space than a rowstore table.

Columnstore Features by Version

In SQL Server 2014, tables with nonclustered columnstore indexes were read-only. You had to drop and recreate the index to update the data – this restriction was removed in SQL Server 2016.

SQL Server 2016 also introduced support for filtered nonclustered columnstore indexes. This allows a predicate condition to filter which rows are included in the index. Use this feature to create an index on only the cold data of an operational workload. This will greatly reduce the performance impact of having a columnstore index on an online transaction processing (OLTP) table.

Clustered Columnstore Indexes

Clustered columnstore indexes, like their rowstore alternatives, optimize the arrangement of the physical data on disk or in memory. In a columnstore index, this will store all the columns next to each other on disk; the structure of the index dictates how the data is stored on disk.

To reduce the impact of fragmentation and to improve performance, a clustered columnstore index might use a deltastore. You can think of a deltastore as a temporary b-tree table with rows that a tuple-mover process moves into the clustered columnstore index at an appropriate time. This moving of row data is performed in the background. When querying the index, it will automatically combine results from the columnstore and deltastore to ensure that the query receives the correct results.

A clustered columnstore index has the following characteristics:

- It includes all of the columns in the table.
- It does not store the columns in a sorted order, but optimizes storage for compression and performance.
- It can be updated.

- Clustered columnstore indexes
 - Must contain all columns
 - Optimize data for storage and performance
 - Row based indexes can be added
 - Cannot be filtered

New features

You can now have a nonclustered row index on top of a clustered columnstore index, making it possible to have efficient table seeks on an underlying columnstore. You can also enforce a primary key constraint by using a unique rowstore index.

SQL Server 2016 introduced columnstore indexes on memory optimized tables—the most relevant use being for real-time operational analytical processing.

SQL Server 2017 introduced support for non-persisted computed columns in nonclustered columnstore indexes.



Note: You cannot include persisted computed columns, and you cannot create nonclustered indexes on computer columns.

For further information, see:

 **Columnstore Indexes for Real-Time Operational Analytics**

<http://aka.ms/bpntqk>

Demonstration: The Benefits of Using Columnstore Indexes

In this demonstration, you will see how to create a columnstore index.

Demonstration Steps

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **D:\Demofiles\Mod07\Setup.cmd** as an administrator to revert any changes.
3. In the **User Account Control** dialog box, click **Yes**. When the script completes, press any key to close the window.
4. On the taskbar, click **Microsoft SQL Server Management Studio**.
5. In the **Connect to Server** window, in the **Server name** box, type **MIA-SQL**. Ensure **Windows Authentication** is selected in the **Authentication** box, and then click **Connect**.
6. On the **File** menu, point to **Open**, click **File**.
7. In the **Open File** dialog box, navigate to **D:\Demofiles\Mod07\Demo\Demo.sql** script file, and then click **Open**.
8. Select the code under the **Step 1** comment, and then click **Execute**.
9. Select the code under the **Step 2** comment, and then click **Execute**.
10. Select the code under the **Step 3** comment, and then click **Execute**.
11. Select the code under the **Step 1** comment, and then click **Execute**.
12. Close Microsoft SQL Server Management Studio without saving changes.

Categorize Activity

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

Items	
1	Perform the best when seeking for specific data.
2	A high degree of compression is possible, due to data being of the same category.
3	Can greatly improve the performance of database queries.
4	Implemented as a b-tree index structure.
5	Perform best when aggregating data.
6	Can be stored in memory optimized tables.

Category 1	Category 2	Category 3
Rowstore Index	Columnstore Index	Applies to both types of index

Lesson 2

Creating Columnstore Indexes

This lesson shows you the techniques required to create columnstore indexes on a table. You will see how to quickly create indexes by using Transact-SQL, or the SQL Server Management Studio user interface.

Lesson Objectives

After completing this lesson, you will be able to:

- Create nonclustered columnstore indexes.
- Create clustered columnstore indexes.
- Create tables that use both columnstore and rowstore indexes.

Creating a Nonclustered Columnstore Index

You can create a nonclustered columnstore index by using a Transact-SQL statement or by using SQL Server Management Studio (SSMS).

Transact-SQL

To create a nonclustered columnstore index, use the **CREATE NONCLUSTERED COLUMNSTORE INDEX** statement, as shown in the following code example:

- Create columnstore indexes using Transact-SQL
 - CREATE NONCLUSTERED COLUMNSTORE INDEX
- Create columnstore indexes using SSMS
- Databases
 - AdventureWorksDW
 - Tables
 - FactFinance
 - Right-click Indexes
 - New index
 - Nonclustered Columnstore Index

Creating a Nonclustered Columnstore Index

```
CREATE NONCLUSTERED COLUMNSTORE INDEX NCCSIX_FactInternetSales
ON FactInternetSales (
    OrderQuantity
    ,UnitPrice
    ,DiscountAmount);
```

A nonclustered index does not need to include all the columns from the underlying table. In the preceding example, only three columns are included in the index.

You can also restrict nonclustered indexes to a subset of the rows contained in a table:

Example of a filtered nonclustered columnstore index

```
CREATE NONCLUSTERED COLUMNSTORE INDEX NCCSIX_FactInternetSales
ON FactInternetSales (
    OrderQuantity
    ,UnitPrice
    ,DiscountAmount);
WHERE ShipDate < '2013-01-01';
```

The business reason for wanting to limit a columnstore index to a subset of rows is that it's possible to use a single table for both OLTP and analytical processing. In the preceding example, the index supports analytical processing on historical orders that shipped before 2013.

SQL Server Management Studio

You can also use Object Explorer in SSMS to create columnstore indexes:

1. In **Object Explorer**, expand **Databases**.
2. Expand the required **Database**; for example, **AdventureWorksDW**.
3. Expand **Tables**, and then expand the required table; for example, **FactFinance**.
4. Right-click **Indexes**, point to **New Index**, and then click **Nonclustered Columnstore Index**.
5. Add at least one column to the index, and then click **OK**.

Creating a Clustered Columnstore Index

Similar to the nonclustered columnstore indexes, a clustered columnstore index is created by using Transact-SQL or SSMS. The main difference between the declaration of a clustered index and nonclustered index is that the former must contain all the columns in the table being indexed.

- Create a clustered columnstore index
- CREATE CLUSTERED COLUMNSTORE INDEX
- SSMS
 - New Index
 - Clustered Columnstore Index

Transact-SQL

To create a clustered columnstore index, use the CREATE CLUSTERED COLUMNSTORE INDEX statement as shown in the following code example:

Creating a Clustered Columnstore Index

```
CREATE CLUSTERED COLUMNSTORE INDEX CCSIX_FactSalesOrderDetails
ON FactSalesOrderDetails;
```

An optional parameter on a CREATE statement for a clustered index is DROP_EXISTING. You can use this to rebuild an existing clustered columnstore index or to convert an existing rowstore table into a columnstore table.

 **Note:** To use the DROP_EXISTING option, the new columnstore index must have the same name as the index it is replacing.

The following example creates a clustered rowstore table, and then converts it into a clustered columnstore table:

Converting a Rowstore Table to a Columnstore Table

```

CREATE TABLE ExampleFactTable (
    ProductKey [int] NOT NULL,
    OrderDateKey [int] NOT NULL,
    DueDateKey [int] NOT NULL,
    ShipDateKey [int],
    CostPrice [money] NOT NULL);
GO
CREATE CLUSTERED INDEX CCI_ExampleFactTable ON ExampleFactTable (ProductKey);
GO
CREATE CLUSTERED COLUMNSTORE INDEX CCI_ExampleFactTable ON ExampleFactTable
WITH (DROP_EXISTING = ON);
GO

```

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
Unable to create columnstore index on an Azure SQL Database.	Ensure you are using at least V12 of an Azure SQL Database. The pricing tier of the database also has to be a minimum of Premium.

SQL Server Management Studio

You can also create a clustered columnstore index by using SSMS:

1. In **Object Explorer**, expand **Databases**.
2. Expand the required **Database**; for example, **AdventureWorksDW**.
3. Expand **Tables**, and then expand the required table; for example, **FactFinance**.
4. Right-click **Indexes**, point to **New Index**, and then click **Clustered Columnstore Index**.
5. Click **OK**.

 **Note:** You don't need to select columns to create a clustered columnstore index, because all the columns of a table must be included.

Creating a Clustered Columnstore Table with Primary and Foreign Keys

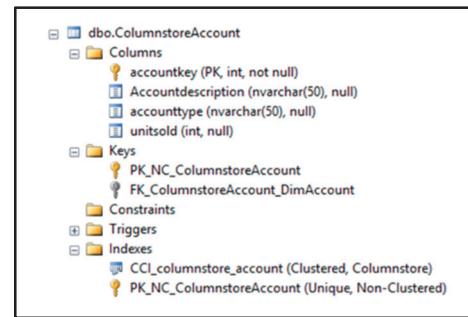
In SQL Server 2014, clustered columnstore indexes are limited in several ways, one of the most significant being that you cannot have any other index on the clustered columnstore table. This essentially means that, in SQL Server 2014, you cannot have a primary key, foreign keys, or unique value constraints, on a clustered columnstore table.

These limitations were removed in SQL Server 2016—these features are all supported in SQL Server 2016 and SQL Server 2017.

This is an example of creating a table with both a primary key and a clustered columnstore index:

Create a table with a primary key and columnstore index

```
CREATE TABLE ColumnstoreAccount (
    accountkey INT NOT NULL,
    Accountdescription NVARCHAR(50),
    accounttype NVARCHAR(50),
    unitsold INT,
    CONSTRAINT PK_NC_ColumnstoreAccount PRIMARY KEY NONCLUSTERED(accountkey ASC),
    INDEX CCI_columnstore_account CLUSTERED COLUMNSTORE);
```



After you create the table, you can add a foreign key constraint:

Add a foreign key constraint

```
ALTER TABLE ColumnstoreAccount WITH CHECK ADD CONSTRAINT FK_ColumnstoreAccount_DimAccount
FOREIGN KEY(AccountKey)
REFERENCES DimAccount (AccountKey)

ALTER TABLE ColumnstoreAccount CHECK CONSTRAINT FK_ColumnstoreAccount_DimAccount
GO
```

Check the table—it shows that two indexes and two keys exist:

- **CCI_columnstore_account**: a clustered columnstore index.
- **PK_NC_ColumnstoreAccount**: a unique nonclustered rowstore index.
- **FK_ColumnstoreAccount_DimAccount**: a foreign key to the DimAccount table.
- **PK_NC_ColumnstoreAccount**: the primary key.

The previous Transact-SQL results in a columnstore index with a nonclustered index that enforces a primary key constraint on both indexes.

Demonstration: Creating Columnstore Indexes Using SQL Server Management Studio

In this demonstration, you will see how to:

- Create a nonclustered columnstore index using SSMS.
- Create a clustered columnstore index using SSMS.

Demonstration Steps

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. On the taskbar, click **Microsoft SQL Server Management Studio**.
3. In the **Connect to Server** window, in the **Server name** box, type **MIA-SQL**. Ensure that **Windows Authentication** is selected in the **Authentication** box, and then click **Connect**.
4. In Object Explorer, expand **Databases**, expand **AdventureWorksDW**, expand **Tables**, and then expand **dbo.AdventureWorksDWBuildVersion**.
5. Right-click **Indexes**, point to **New Index**, and then click **Clustered Columnstore Index**.
6. In the **New Index** dialog box, click **OK** to create the index.
7. In Object Explorer, expand **Indexes** to show the new clustered index.
8. In Object Explorer, expand **dbo.FactResellersSales**.
9. Right-click **Indexes**, point to **New Index**, and then click **Non-Clustered Columnstore Index**.
10. In the **Columnstore columns** table, click **Add**.
11. Select the **SalesOrderNumber**, **UnitPrice**, and **ExtendedAmount** check boxes, and then click **OK**.
12. In the **New Index** dialog box, click **OK**.
13. In Object Explorer, expand **Indexes** to show the created nonclustered index.
14. Close Microsoft SQL Server Management Studio without saving.

Question: How will you create your indexes in a database—with SSMS or Transact-SQL?

Lesson 3

Working with Columnstore Indexes

When working with columnstore indexes, you should consider fragmentation and how SQL Server processes the insertion of data into the index. From SQL Server 2016, columnstore tables can be created in memory. This makes real-time operational analytics possible.

Lesson Objectives

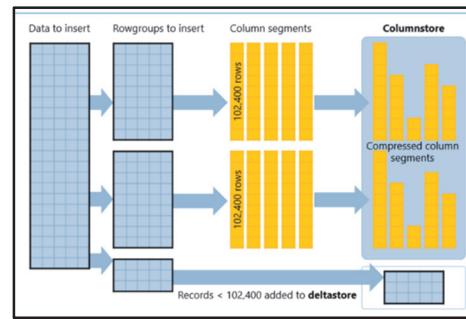
After completing this lesson, you will be able to:

- Efficiently add data into a columnstore table.
- Check the fragmentation of an index and choose the best approach to resolving the fragmentation.
- Create a memory optimized table to support real-time operational analytics.

Managing Columnstore Indexes

Columnstore indexes have similar management considerations as rowstore indexes—however, special consideration must be given to DML operations.

For data to be manipulated and changed in a columnstore table, SQL Server manages a deltastore. The deltastore collects up to 1,048,576 rows before compressing them into the compressed rowgroup, and then marking that rowgroup as closed. The tuple-mover background process then adds the closed rowgroup back into the columnstore index.



Bypassing the Deltastore

To ensure that data is inserted directly into the columnstore, you should load the data in batches of between 102,400 and 1,048,576 rows. This bulk data loading is performed through normal insertion methods, including using the bcp utility, SQL Server Integration Services, and Transact-SQL insert statements from a staging table.

When bulk loading data, you have the following options for optimizations:

- **Parallel Load:** perform multiple concurrent bulk imports (bcp or bulk insert), each loading separate data.
- **Log Optimization:** the bulk load will be minimally logged when the data is loaded into a compressed rowgroup. Minimal logging is not available when loading data with a batch size of less than 102,400 rows.

Index Fragmentation

When it comes to managing, columnstore indexes are no different to rowstore indexes. The utilities and techniques used to keep an index healthy are the same.

Over time, and after numerous DML operations, indexes can become fragmented and their performance degrades. SSMS provides Transact-SQL commands and user interface elements to help you manage this fragmentation.

- Use `sys.dm_db_index_physical_stats` to determine index fragmentation
- Reorganize
 - Fragmentation between 5% and 30%
- Rebuild
 - Fragmentation greater than 30%

SQL Server Management Studio

You can examine the level of fragmentation by following these steps:

1. In Object Explorer, expand **Databases**.
2. Expand the required **Database**; for example, **AdventureWorksDW**.
3. Expand **Tables**, and then expand the required table; for example, **FactFinance**.
4. Expand **Indexes**, right-click on the desired index, and in the context menu, click **Properties**.
5. In the **Select a page** panel, click **Fragmentation**.

Transact-SQL

SQL Server provides dynamic management views and functions that make it possible for a database administrator to inspect and review the health of indexes.

One of these functions is **`sys.dm_db_index_physical_stats`** that can be run against all the databases on a server, a specific table in a database, or even a specific index.

The following code sample shows a useful query that joins the results from the **`sys.dm_db_index_physical_stats`** view with the system index table, and then returns the fragmentation and names of the indexes for a specific database:

Show indexes with fragmentation greater than five percent for a specific database

```
SELECT DB_NAME(database_id) AS 'Database'
      , OBJECT_NAME(dm.object_id) AS 'Table'
      , si.name AS 'Index'
      , dm.index_type_desc
      , dm.avg_fragmentation_in_percent
  FROM sys.dm_db_index_physical_stats
  (DB_ID(N'AdventureworksDW'), NULL, NULL, NULL, 'DETAILED') dm
  JOIN sys.indexes si ON si.object_id = dm.object_id AND si.index_id = dm.index_id
 WHERE avg_fragmentation_in_percent > 5
 ORDER BY avg_fragmentation_in_percent DESC;
```

When you identify that an index requires maintenance, there are two options available in SQL Server: you can either rebuild or reorganize it. The previous guidance for this was:

- If the fragmentation is between five percent and 30 percent: **Reorganize**.
- If the fragmentation is greater than 30 percent: **Rebuild**.

However, the reorganizing of columnstore indexes is enhanced in SQL Server 2016 and SQL Server 2017; therefore, it is rarely necessary to rebuild an index.

Use the following Transact-SQL to reorganize an index:

Transact-SQL to reorganize an index online

```
ALTER INDEX CCI_columnstore_account ON ColumnstoreAccount
REORGANIZE WITH (COMPRESS_ALL_ROW_GROUPS = ON)
GO

ALTER INDEX CCI_columnstore_account ON ColumnstoreAccount REORGANIZE
GO
```

The first statement in this code sample adds deltastore rowgroups into the columnstore index. Using the COMPRESS_ALL_ROW_GROUPS option forces all open and closed rowgroups into the index, in a similar way to rebuilding an index. After the query adds these deltastore rowgroups to the columnstore, the second statement then combines these, possibly smaller, rowgroups into one or more larger rowgroups. With a large number of smaller rowgroups, performing the reorganization a second time will improve the performance of queries against the index. Using these statements in SQL Server 2016 or SQL Server 2017 means that, in most situations, you no longer need to rebuild a columnstore index.

 **Note:** Rebuilding an index will mean SQL Server can move the data in the index between segments to achieve better overall compression. If a large number of rows are deleted, and the index fragmentation is more than 30 percent, rebuilding the index may be the best option, rather than reorganizing.

 **For more information on all the available views and functions, see:**

<http://aka.ms/vched5>

Columnstore Indexes and Memory Optimized Tables

In SQL Server 2016 and SQL Server 2017, you can now have both a clustered columnstore index and rowstore index on an in-memory table. The combination of these indexes on a table enables real-time operational analytics and all of its associated benefits, including:

- **Simplicity.** No need to implement an ETL process to move data into reporting tables. Analytics can run directly on the operational data.
- **Reduced costs.** Removes the need to develop and support ETL processes. The associated hardware and infrastructure to support the ETL are no longer necessary.
- **Zero data latency.** Data is analyzed in real time. There are no background or schedule processes to move data to enable analytics to be completed.

In-memory columnstore tables

- The index has to be declared at runtime
- Tables can be up to 2 TB in size
- Can be combined with rowstore index
- Enable real-time operational analytics

Memory Optimization Advisor can be used to support moving a table from being disk based to being memory-optimized.

The combination of indexes enables analytical queries to run against the columnstore index and OLTP operations to run against the OLTP b-tree indexes. The OLTP workloads will continue to perform, but you may incur some additional overhead when maintaining the columnstore index.

 **For more information on Real-Time Operational Analytics, see:**

<http://aka.ms/bpntqk>

As with other similar in-memory tables, you must declare the indexes on memory optimized columnstore tables at creation. To support larger datasets—for example, those used in data warehouses—the size of in-memory tables has increased from a previous limit of 256 GB to 2 TB in SQL Server 2016 and SQL Server 2017.

The Transact-SQL to create an in-memory table is simple. Add WITH (MEMORY_OPTIMIZED = ON) at the end of a table declaration.

Transact-SQL to create a columnstore in-memory table.

```
CREATE TABLE InMemoryAccount (
    accountkey int NOT NULL,
    accountdescription nvarchar (50),
    accounttype nvarchar(50),
    unitsold int,
        CONSTRAINT [PK_NC_InMemoryAccount] PRIMARY KEY NONCLUSTERED([accountkey]
ASC),
    INDEX CCI_InMemoryAccount CLUSTERED COLUMNSTORE)
    WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA)
GO
```

 **Note:** The above Transact-SQL will not work on a database without a memory optimized file group. Before creating any memory optimized tables, the database must have a memory optimized file group associated with it.

The following is an example of the code required to create a memory optimized filegroup:

Adding a memory optimized filegroup

```
ALTER DATABASE AdventureWorksDW
ADD FILEGROUP AdventureWorksDW_Memory_Optimized_Data CONTAINS MEMORY_OPTIMIZED_DATA
GO

ALTER DATABASE AdventureWorksDW ADD
FILE (name='AdventureworksDW_MOD', filename='D:\AdventureworksDW_MOD')
TO FILEGROUP AdventureWorksDW_Memory_Optimized_Data
GO
```

You can alter and join this new in-memory table in the same way as its disk based counterpart. However, you should use caution when altering an in-memory table, because this is an offline task. There should be twice the memory available to store the current table—a temporary table is used before being switched over when it has been rebuilt.

Depending on the performance requirements, you can control the durability of the table by using:

- **SCHEMA_ONLY:** creates a nondurable table.
- **SCHEMA_AND_DATA:** creates a durable table; this is the default if no option is supplied.

If SQL Server restarts, a schema-only table loses all of its data. Temporary tables, or transient data, are examples of where you might use a nondurable table. Because SCHEMA_ONLY durability avoids both transaction logging and checkpoints, I/O operations can be reduced significantly.

Introduced in SQL Server 2014, the Memory Optimization Advisor is a GUI tool inside SSMS. The tool will analyze an existing disk based table and warn if there are any features of that table—for example, an unsupported type of index—that aren't possible on a memory-optimized table. It can then migrate the data contained in the disk based table to a new memory-optimized table. The Memory Optimization Advisor is available on the context menu of any table in Management Studio.

Check Your Knowledge

Question	
When would you consider converting a rowstore table, containing dimension data in a data warehouse, to a columnstore table?	
Select the correct answer.	
	When mission critical analytical queries join one or more fact tables to the dimension table, and those fact tables are columnstore tables.
	When the data contained in the dimension table has a high degree of randomness and uniqueness.
	When the dimension table has very few rows.
	When the dimension table has many millions of rows, with columns containing small variations in data.
	It is never appropriate to convert a dimension table to a columnstore table.

Lab: Using Columnstore Indexes

Scenario

Adventure Works has created a data warehouse for analytics processing of its current online sales business. Due to large business growth, existing analytical queries are no longer performing as required. Disk space is also becoming more of an issue.

You have been tasked with optimizing the existing database workloads and, if possible, reducing the amount of disk space being used by the data warehouse.

Objectives

After completing this lab, you will be able to:

- Create clustered and nonclustered columnstore indexes.
- Examine an execution plan to check the performance of queries.
- Convert disk based tables into memory optimized tables.

Lab Setup

Estimated Time: 45 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Dropping and recreating indexes can take time, depending on the performance of the lab machines.

Exercise 1: Create a Columnstore Index on the FactProductInventory Table

Scenario

You plan to improve the performance of the **AdventureWorksDW** data warehouse by using columnstore indexes. You need to improve the performance of queries that use the **FactProductInventory** tables without causing any database downtime, or dropping any existing indexes. Disk usage for this table is not an issue.

You must retain the existing indexes on the **FactProductInventory** table, and ensure you do not impact current applications by any alterations you make.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Examine the Existing Size of the FactProductInventory Table and Query Performance
3. Create a Columnstore Index on the FactProductInventory Table

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab07\Starter** folder, run **Setup.cmd** as Administrator.

► **Task 2: Examine the Existing Size of the FactProductInventory Table and Query Performance**

1. In SQL Server Management Studio, in the **D:\Labfiles\Lab07\Starter** folder, open the **Query FactProductInventory.sql** script file.
2. Configure SQL Server Management Studio to include the actual execution plan.
3. Execute the script against the **AdventureWorksDW** database. Review the execution plan, making a note of the indexes used, the execution time, and disk space used.

► **Task 3: Create a Columnstore Index on the FactProductInventory Table**

1. Based on the scenario for this exercise, decide whether a clustered or nonclustered columnstore index is appropriate for the **FactProductInventory** table.
2. Create the required columnstore index. Re-execute the query to verify that the new columnstore index is used, along with existing indexes.
3. What, if any, are the disk space and query performance improvements?

Results: After completing this exercise, you will have created a columnstore index and improved the performance of an analytical query. This will have been done in real time without impacting transactional processing.

Exercise 2: Create a Columnstore Index on the FactInternetSales Table

Scenario

You need to improve the performance of queries that use the **FactInternetSales** table. The table has also become large and there are concerns over the disk space being used. You can use scheduled downtime to amend the table and its existing indexes.

Due to existing processing requirements, you must retain the foreign keys on the **FactInternetSales** table, but you can add any number of new indexes to the table.

The main tasks for this exercise are as follows:

1. Examine the Existing Size of the FactInternetSales Table and Query Performance
2. Create a Columnstore Index on the FactInternetSales Table

► **Task 1: Examine the Existing Size of the FactInternetSales Table and Query Performance**

1. In SQL Server Management Studio, in the **D:\Labfiles\Lab07\Starter** folder, open the **Query FactInternetSales.sql** script file.
2. Configure SQL Server Management Studio to include the actual execution plan.
3. Execute the script against the **AdventureWorksDW** database. Review the execution plan, making a note of the indexes used, the execution time, and disk space used.

► **Task 2: Create a Columnstore Index on the FactInternetSales Table**

1. Based on the scenario for this exercise, decide whether a clustered or nonclustered columnstore index is appropriate for the **FactInternetSales** table.
2. Create the required columnstore index. Depending on your chosen index, you may need to drop and recreate keys on the table.

3. Re-execute the query to verify that the new columnstore index is used, along with the existing indexes.
4. What, if any, are the disk space and query performance improvements?

Results: After completing this exercise, you will have greatly reduced the disk space taken up by the FactInternetSales table, and improved the performance of analytical queries against the table.

Exercise 3: Create a Memory Optimized Columnstore Table

Scenario

Due to the improved performance and reduced disk space that columnstore indexes provide, you have been tasked with taking the FactInternetSales table from disk and into memory.

The main tasks for this exercise are as follows:

1. Use the Memory Optimization Advisor
2. Enable the Memory Optimization Advisor to Create a Memory Optimized FactInternetSales Table
3. Examine the Performance of the Memory Optimized Table

► Task 1: Use the Memory Optimization Advisor

1. In SQL Server Management Studio, run the **Memory Optimization Advisor** on the **FactInternetSales** table.
2. Note that there are several issues that need to be resolved before the Memory Optimization Advisor can automatically convert the table.

► Task 2: Enable the Memory Optimization Advisor to Create a Memory Optimized FactInternetSales Table

1. Using either SQL Server Management Studio, or Transact-SQL statements, drop all the foreign keys and the clustered columnstore index.
2. Memory optimized tables cannot have more than eight indexes. Choose another three indexes to drop.



Note: Hint: consider the rows being used in the **Query FactInternetSales.sql** to guide your decision.

3. Use **Memory Optimization Advisor** on the **FactInternetSales** table.
4. Instead of running the migration with the wizard, script the results for the addition of a columnstore index.



Note: The Memory Optimization Advisor won't suggest columnstore indexes as they are not applicable in all situations. Therefore, these have to be added manually.

5. Note the statements to create a memory optimized filegroup, and the code to copy the existing data.
6. Add a clustered columnstore index to the create table script.
7. Run the edited Transact-SQL.

► **Task 3: Examine the Performance of the Memory Optimized Table**

1. In SQL Server Management Studio, in the **D:\Labfiles\Lab07\Starter** folder, open the **Query FactProductInventory.sql** script file.
2. Configure SQL Server Management Studio to include the actual execution plan.
3. Execute the script against the **AdventureWorksDW** database, and then review the disk space used the execution plan.

Results: After completing this exercise, you will have created a memory optimized version of the **FactInternetSales** disk based table, using the Memory Optimization Advisor.

Question: Why do you think the disk space savings were so large for the disk based clustered columnstore index?

Module Review and Takeaways



Best Practice: Introduced in SQL Server 2012, columnstore indexes are used in large data warehouse solutions by many organizations. This module highlighted the benefits of using these indexes on large datasets; the improvements made to columnstore indexes in SQL Server 2016; and the considerations needed to use columnstore indexes effectively in your solutions.

Module 8

Designing and Implementing Views

Contents:

Module Overview	8-1
Lesson 1: Introduction to Views	8-2
Lesson 2: Creating and Managing Views	8-9
Lesson 3: Performance Considerations for Views	8-18
Lab: Designing and Implementing Views	8-22
Module Review and Takeaways	8-25

Module Overview

This module describes the design and implementation of views. A view is a special type of query—one that is stored and can be used in other queries—just like a table. With a view, only the query definition is stored on disk; not the result set. The only exception to this is indexed views, when the result set is also stored on disk, just like a table.

Views simplify the design of a database by providing a layer of abstraction, and hiding the complexity of table joins. Views are also a way of securing your data by giving users permissions to use a view, without giving them permissions to the underlying objects. This means data can be kept private, and can only be viewed by appropriate users.

Objectives

After completing this module, you will be able to:

- Understand the role of views in database design.
- Create and manage views.
- Understand the performance considerations with views.

Lesson 1

Introduction to Views

After completing this lesson, you will be able to:

- Describe a view.
- Describe the different types of views in SQL Server®.
- Explain the benefits of using views.
- Work with dynamic management views.
- Work with other types of system view.

Lesson Objectives

In this lesson, you will explore the role of views in the design and implementation of a database. You will also investigate the system views supplied with Microsoft® SQL Server data management software.

A view is a named SELECT query that produces a result set for a particular purpose. Unlike the underlying tables that hold data, a view is not part of the physical schema. Views are dynamic, virtual tables that display specific data from tables.

The data returned by a view might filter the table data, or perform operations on the table data to make it suitable for a particular need. For example, you might create a view that produces data for reporting, or a view that is relevant to a specific group of users. The effective use of views in database design improves performance, security, and manageability of data.

In this lesson, you will learn about views, the different types of views, and how to use them.

What Is a View?

A view is a stored query expression. The query expression defines what the view will return; it is given a name, and is stored ready for use when the view is referenced. Although a view behaves like a table, it does not store any data. So a view object takes up very little space—the data that is returned comes from the underlying base tables.

Views are defined by using a SELECT statement. They are named—so the definition can be stored and referenced. They can be referenced in SELECT statements; and they can reference other views.

- A view is a stored query expression:
 - It behaves like a table, but the data is stored in the underlying tables
 - It has a name
 - It can be referenced by other queries
 - Filter records by restricting the columns or rows that are returned
- Use views to:
 - Simplify the complex underlying relationship between tables
 - Prevent unauthorized access to data

Filter Data Using Views

Views can filter the base tables by limiting the columns that a view returns. For example, an application might show a drop-down list of employee names. This data could be retrieved from the **Employee** table; however, not all the columns in the **Employee** table might be suitable for including in a selection box. By creating a view, you can limit the returned columns to only those that are necessary, and only those that users are permitted to see. This is known as vertical filtering.

Horizontal filtering limits the rows that a view returns. For example, a **Sales** table might hold details of sales for an organization, but sales staff are only permitted to view sales for their own region. You could create a view that returns only the rows for a particular state or region.

When to Use a View

Views can be used to simplify the way data is presented, hiding complex relationships between tables. Views can also be used to prevent unauthorized access to data. If a user does not have permissions to see salary information, for example, a view can be created that displays data without the salary data. Appropriate groups of users can then be given permissions for that view.

Types of Views

There are two main groups of views: user-defined views that you create and manage in a database, and system views that SQL Server manages.

User-defined Views

You can create three types of user-defined views:

- **Views or standard views.** A view combines data from one or more base tables, or views. Any computations, such as joins or aggregations, are performed during query execution for each query that references the view. This is sometimes called a standard view, or a nonindexed view.
- **Indexed views.** An indexed view stores data by creating a clustered index on the view. By indexing the view, the data is stored on disk and so can be retrieved more quickly in future. This can significantly improve the performance of some queries, including those that aggregate a large number of rows, or where tables are joined and the results stored. If the underlying data changes frequently, however, an indexed view is less likely to be suitable. Indexed views are discussed later in this module.
- **Partitioned views.** Partitioned views join data from one or more tables using the UNION operator. Rows from one table are joined to the rows from one or more other tables into a single view. The columns must be the same, and CHECK constraints on the underlying tables enforce which rows belong to which tables. Local partitioned views include tables from the same SQL Server instance, and distributed partitioned views include tables that are located on different servers.

- User-defined views:
 - Views (sometimes called standard views)
 - Indexed views
 - Partitioned views
 - System views:
 - System catalog views
 - Dynamic management views (DMVs)
 - Compatibility views
 - Information schema views

System Views

In addition, there are different types of system views, including:

- **Dynamic management views (DMVs)** provide dynamic state information, such as data about the current session or the queries that are currently executing.
- **System catalog views** provide information about the state of the SQL Server Database Engine.
- **Compatibility views** are provided for backwards compatibility and replace the system tables used by previous versions of SQL Server.
- **Information schema views** provide internal, table-independent metadata that comply with the ISO standard definition for the INFORMATION_SCHEMA.

Advantages of Views

Views have a number of benefits in your database.

Simplify

Views simplify the complex relationships between tables by showing only relevant data. Views help users to focus on a subset of data that is relevant to them, or that they are permitted to work with. Users do not need to see the complex queries that are often involved in creating the view; they work with the view as if it were a single table.

- Views have many advantages:
 - Create a simplified view of the underlying table relationships
 - Provide data security—users see only what they need to see
 - Can create an interface between underlying data structures and external applications
 - If changes are made to tables, you need to make sure the views still work
 - Simplify reporting by providing data in the correct format

Security

Views provide security by permitting users to see only what they are authorized to see. You can use views to limit the access to certain data sets. By only including data that users are authorized to see, private data is kept private. Views are widely used as a security mechanism by giving users access to data through the view, but not granting permissions to the underlying base tables.

Provide an Interface

Views can provide an interface to the underlying tables for users and applications. This provides a layer of abstraction in addition to backwards compatibility if the base tables change.

Many external applications cannot execute stored procedures or Transact-SQL code, but can select data from tables or views. By creating a view, you can isolate the data that is needed.

Creating a view as an interface makes it easier to maintain backwards compatibility. Providing the view still works, the application will work—even if changes have been made to the underlying schema. For example, if you split a Customer table into two, CustomerGeneral and CustomerCredit, a Customer view can make it appear that the Customer table still exists, allowing existing applications to query the data without modifications.

Format Data for Reporting

Correctly formatted data can be provided to reporting applications, thereby removing the need for complex queries at the application layer.

Reporting applications often need to execute complex queries to retrieve the report data. Rather than embedding this logic in the reporting application, a view can supply the data in the format required by the reporting application.

Dynamic Management Views

Dynamic management views (DMVs) provide information about the internal state of a SQL Server database or server. DMVs, together with dynamic management functions (DMFs), are known as dynamic management objects (DMOs). The key difference between DMVs and DMFs is that DMFs take parameters.

All DMOs have the name prefix **sys.dm_** and are stored in the system schema. As their name implies, they return dynamic results, based on the current state of the object that you are querying.

You can view a complete list of DMVs under the **Views** node for a database in Object Explorer. DMFs are listed by category under the **System Functions** node.

You can use DMOs to view and monitor the internal health and performance of a server, along with aspects of its configuration. They also have an important role in assisting with troubleshooting problems, such as blocking issues, and with performance tuning.

- DMVs return internal SQL Server state information
- DMFs take parameters
- DMVs and DMFs are together known as DMOs
- Use DMOs to:
 - Monitor the health of a SQL Server instance
 - Diagnose problems
 - Tune performance

 **Note:** The schema and data returned by DMVs and DMFs may change in future releases of SQL Server, impacting forward compatibility. It is recommended that you explicitly define the columns of interest in SELECT statements, rather than using SELECT *, to ensure the expected number and order of columns are returned.

Other System Views

In addition to DMOs, SQL Server provides catalog views, compatibility views, and information schema views that you can use to access system information.

Catalog Views

SQL Server exposes information relating to database objects through catalog views. Catalog views provide metadata that describes both user created database objects, and SQL Server system objects. For example, you can use catalog views to retrieve metadata about tables, indexes, and other database objects.

- Catalog views:
 - Views onto internal system metadata
 - Organized into categories, such as object views, schema views, or linked server views
- Compatibility views:
 - Provide backward compatibility for SQL Server 2000 system tables
 - Do not use for new development work
- Information schema views:
 - Comply with the ISO standard

This sample code uses the **sys.tables** catalog view, together with the OBJECTPROPERTY system function, to retrieve all the tables that have IDENTITY columns. The **sys.tables** catalog view returns a row for each user table in the database.

Using the sys.tables Catalog View

```
USE AdventureWorks2016;
GO

SELECT SCHEMA_NAME(schema_id) AS 'Schema', name AS 'Table'
    FROM sys.tables
    WHERE OBJECTPROPERTY(object_id,'TableHasIdentity') = 1
        ORDER BY 'Schema', 'Table';
GO
```

Catalog views are categorized by their functionality. For example, object catalog views report on object metadata.

Some catalog views inherit from others. For example, **sys.views** and **sys.tables** inherit from **sys.objects**. That is, **sys.objects** returns metadata for all user-defined database objects, whereas **sys.views** returns a row for each user-defined view, and **sys.tables** returns a row for each user-defined table.

 **Note:** Catalog views are updated with information on the new releases of SQL Server. Use `SELECT * FROM sys.objects`—for example, to get all the data from a catalog view.

For more information about catalog views and the different categories of catalog views, see Microsoft Docs:

 **System Catalog Views (Transact-SQL)**

<http://aka.ms/lfswev>

Compatibility Views

Before catalog views were introduced in SQL Server 2005, you would use system tables to retrieve information about internal objects. For backward compatibility, a set of compatibility views are available so that applications continue to work. However, compatibility views only expose information relevant to SQL Server 2000. For new development work, you should use the more up-to-date catalog views.

For more information on compatibility views, see Microsoft Docs:

 **Compatibility Views (Transact-SQL)**

<http://aka.ms/w0a477>

Information Schema Views

Information schema views comply with ISO standards for SQL. This was developed because different database vendors use different methods of storing and accessing metadata. The ISO standard provides a common format for all database products.

Examples of commonly used INFORMATION_SCHEMA views include:

Using INFORMATION_SCHEMA Views

```
SELECT * FROM INFORMATION_SCHEMA.TABLES;
SELECT * FROM INFORMATION_SCHEMA.PARAMETERS;
SELECT * FROM INFORMATION_SCHEMA.COLUMNS;
SELECT * FROM INFORMATION_SCHEMA.COLUMN_PRIVILEGES;
SELECT * FROM INFORMATION_SCHEMA.CHECK_CONSTRAINTS;
```

For more details about information schema views, see Microsoft Docs:

Information Schema Views (Transact-SQL)

<http://aka.ms/yh3gmm>

Demonstration: Querying Catalog Views and DMVs

In this demonstration, you will see how to:

- Query catalog views and INFORMATION_SCHEMA views.
- Query DMVs.

Demonstration Steps

Query System Views and Dynamic Management Views

1. Ensure that the **MT17B-WS2016-NAT**, **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **AdventureWorks\Student** with the password **Pa55w.rd**.
2. Start SQL Server Management Studio.
3. In the **Connect to Server** dialog box, in the **Server name** box, type the name of the Azure server you created before the course started; for example, <**servername**>.database.windows.net.
4. In the **Authentication** list, click **SQL Server Authentication**.
5. In the **Login** box, type **Student**, in the **Password** box, type **Pa55w.rd**, and then click **Connect**.
6. On the **File** menu, point to **Open**, and then click **File**.
7. In the **Open File** dialog box, navigate to **D:\Demofiles\Mod08**, click **Mod_08_Demo_1A.sql**, and then click **Open**.
8. On the toolbar, in the **Available Databases** list, click **AdventureWorksLT**.
9. Select the code under the **Step 2 - Query sys.views** comment, and then click **Execute**.
10. Select the code under the **Step 3 - Query sys.tables** comment, and then click **Execute**.
11. Select the code under the **Step 4 - Query sys.objects** comment, and then click **Execute**.
12. Select the code under the **Step 5 - Query information_schema.tables** comment, and then click **Execute**.
13. In Object Explorer, expand **Databases**, expand **AdventureWorksLT**, expand **Views**, and then expand **System Views**. Note the system views and user-defined views.
14. Select the code under the **Step 6 - Query sys.dm_exec_connections** comment, and then click **Execute**.

15. Select the code under the **Step 7 - Query sys.dm_exec_sessions** comment, and then click **Execute**.
16. Select the code under the **Step 8 - Query sys.dm_exec_requests** comment, and then click **Execute**.
17. Select the code under the **Step 9 - Query sys.dm_exec_query_stats** comment, and then click **Execute**.
18. Select the code under the **Step 10 - Modify the query to add a TOP(20) and an ORDER BY** comment, and then click **Execute**.
19. Leave SSMS open for the next demonstration.

Check Your Knowledge

Question
What is the purpose of a view?
Select the correct answer.
<input type="checkbox"/> To list all the DMVs available in SQL Server.
<input type="checkbox"/> To present relevant information to users and hide complexity.
<input type="checkbox"/> To remove the need for any security in a SQL Server database.
<input type="checkbox"/> To encrypt data in certain tables.
<input type="checkbox"/> To make tables faster to update.

Lesson 2

Creating and Managing Views

In this lesson, you will learn how to create a view, in addition to how to alter and drop a view. You will learn about how views, and the objects on which they are based, have owners. You will also learn how to find information about existing views, and work with updateable views. You will find information about existing views, and how to obfuscate the definition of views.

Lesson Objectives

After completing this lesson, you will be able to:

- Create a view.
- Drop a view.
- Alter a view.
- Explain the concept of ownership chains, and how it applies to views.
- Retrieve information about views.
- Work with updateable views.
- Obfuscate view definitions.

Create a View

CREATE VIEW

To create a new view, use the CREATE VIEW command. At its simplest, you create a view by giving it a name and writing a SELECT statement. To show only the names of current employees, you can create an employee list view that includes only Title, FirstName, MiddleName, and LastName. As with tables, column names must be unique. If you are using an expression, it must have an alias.

Create a new view.

Use the CREATE VIEW command

```
CREATE VIEW Person.vwEmployeeList AS
SELECT Title, FirstName, MiddleName, LastName
    FROM person.person
    INNER JOIN HumanResources.Employee
        ON Person.BusinessEntityID = Employee.BusinessEntityID
        WHERE Employee.CurrentFlag = 1;
GO
```

- Use the CREATE VIEW statement to create a new view
- View attributes
 - WITH ENCRYPTION
 - WITH SCHEMABINDING
 - WITH VIEW_METADATA
 - WITH CHECK option
 - Ensures new records conform to the view definition

 **Best Practice:** It is good practice to prefix the name of your view with vw; for example, vwEmployeeList. Although database developers differ in their naming conventions, most would agree that it is beneficial to be able to see clearly which objects are tables, and which are views.

Within the SELECT statement, you can reference other views instead of, or in addition to, base tables. Up to 32 levels of nesting are permitted. However, the practice of deeply nesting views quickly becomes difficult to understand and debug. Any performance problems can also be difficult to fix.

In a view definition, you cannot use either the INTO keyword or the OPTION clause. Also, because view definitions are permanent objects within the database, you cannot reference a temporary table or table variable. Views have no natural output order so queries that access the views need to specify the order for the returned rows.

 **Note:** You cannot guarantee ordered results in a view definition. Although you can use the ORDER BY clause, it is only used to determine the rows returned by the TOP, OFFSET, or FOR XML clauses. It does not determine the order of the returned rows.

Once created, views behave much like a table; for example, you can query the view just as you would query a table.

View Attributes

There are three view attributes:

WITH ENCRYPTION

The WITH ENCRYPTION attribute obfuscates the view definition in catalog views where the text of CREATE VIEW is held. It also prevents the view definition being displayed from Object Explorer. WITH ENCRYPTION also stops the view from being included in SQL Server replication.

WITH SCHEMABINDING

You can specify the WITH SCHEMABINDING option to stop the underlying table(s) being changed in a way that would affect the view definition. Indexed views must use the WITH SCHEMABINDING option.

WITH VIEW_METADATA

The WITH VIEW_METADATA attribute determines how SQL Server returns information to ODBC drivers and the OLE DB API. Normally, metadata about the underlying tables is returned, rather than metadata about the view. This is a potential security loophole—by using WITH VIEW_METADATA, the metadata returned is the view name, and not the underlying table names.

 **Note:** The WITH ENCRYPTION attribute does not encrypt the data being returned by the view; it only encrypts the view definition stored in catalog views, such as **sys.syscomments**.

The WITH CHECK Option

The WITH CHECK option is used if you want to ensure that, when updates are made to underlying tables through the view, they comply with any filtering that the view defines. For example, consider a view that includes a WHERE clause to return only employees working in the Sales department. This would prevent you from adding a new record to the underlying table for an employee working in Finance.

After you have created a view, you can work with it as if it were a table.

Querying Views

```
SELECT *
FROM Person.vwEmployeeList;
GO
```

You can join a view to a table as if it were a table.

Joining Views to Tables

```
SELECT E.FirstName, E.LastName, H.NationalIDNumber
FROM Person.vwEmployeeList E
INNER JOIN HumanResources.Employee H
ON E.BusinessEntityID = H.BusinessEntityID;
GO
```

Drop a View

To remove a view from a database, use the DROP VIEW statement. This removes the definition of the view, and all associated permissions.

Use the DROP VIEW statement to delete a view from the database.

DROP VIEW

```
DROP VIEW Person.vwEmployeeList;
GO
```

- To remove a view from the database, use the DROP VIEW statement
- Also drops associated permissions
- Multiple views can be dropped in a single statement
 - Comma-delimited list of views to be deleted

Even if a view is recreated with exactly the same name as a view that has been dropped, permissions that were formerly associated with the view are removed.

 **Best Practice:** Keep database documentation up to date, including the purpose for each view you create, and where they are used. This will help to identify views that are no longer required. These views can then be dropped from the database. Keeping old views that have no use makes database administration more complex, and adds unnecessary work, particularly at upgrade time.

If a view was created using the WITH SCHEMABINDING option, you will need to either alter the view, or drop the view, if you want to make changes to the structure of the underlying tables.

You can drop multiple views with one comma-delimited list, as shown in the following example:

Dropping Multiple Views

```
DROP VIEW Person.vwEmployeeList, Person.vwSalaries;
GO
```

Alter a View

After a view is defined, you can modify its definition without dropping and recreating the view.

The ALTER VIEW statement modifies a previously created view. This includes indexed views, which are discussed in the next lesson. One advantage of ALTER VIEW is that any associated permissions are retained.

For example, if you want to remove the Title field from the Person.vwEmployeeList view, and add the BusinessEntityID.

For example, use the following code to remove the Title field from the Person.vwEmployeeList view, and add the BusinessEntityID column to the view:

- To alter an existing view definition, use the ALTER VIEW statement
- Does not alter associated permissions

Altering a View

```
ALTER VIEW Person.vwEmployeeList AS
SELECT Person.BusinessEntityID, FirstName, MiddleName, LastName
    FROM Person.Person
    INNER JOIN HumanResources.Employee
    ON Person.BusinessEntityID = Employee.BusinessEntityID
    WHERE Employee.CurrentFlag = 1;
GO
```

Ownership Chains and Views

An ownership chain refers to database objects that reference each other in a chain. Each database object has an owner—SQL Server compares the owner of an object to the owner of the calling object.

When you are querying a view, you must have an unbroken chain of ownership from the view to the underlying tables. Users who execute a query on a view, and who have permissions on the underlying tables, will always be allowed to query the view.

- Views and tables that have the same owner:
 - Another user can be given access to the view—even if they don't have permissions on the table
 - This enables views to filter information from the underlying table
- If a view and underlying tables have different owners:
 - Another user will not have access to the view
 - Applies even if the owner of the view has access to the tables

Checking Permissions in an Ownership Chain

Views are often used to provide a layer of security between the underlying tables, and the data that users see. Access is allowed to a view, but not the underlying tables. For this to function correctly, an unbroken ownership chain must exist.

For example, John has no access to a table that Nupur owns. If Nupur creates a view or stored procedure that accesses the table and gives John permission to the view, John can then access the view and, through it, the data in the underlying table. However, if Nupur creates a view or stored procedure that accesses a table that Tim owns and grants John access to the view or stored procedure, John would not be able to

use the view or stored procedure—even if Nupur has access to Tim's table, because of the broken ownership chain. Two options are available to correct this situation:

- Tim could own the view or stored procedure instead of Nupur.
- John could be granted permission to the underlying table.

It is not always desirable to grant permissions to the underlying table, and views are often used as a way of limiting access to certain data.

Ownership Chains vs. Schemas

SQL Server 2005 introduced the concept of schemas. At that point, the two-part naming for objects changed from owner.object to schema.object. All objects still have owners, including schemas. Security is simplified if schema owners also own the objects that are contained in the schemas.

Sources of Information about Views

After views have been created, there are a number of ways you can find information about them, including their definition.

SSMS Object Explorer

SQL Server Management Studio (SSMS) Object Explorer lists system views and user defined views under the database that contains them. Here you can also see the columns, triggers, indexes, and statistics that are defined on the view.

You can use the standard SSMS scripting functionality from Object Explorer to script the view to a defined location. However, if the view was created using the WITH ENCRYPTION option, you cannot script the definition.

You can also use the **Script View as** menu option to alter, drop, or select from a view. In each case, the script is opened in a new query window for you to inspect, and then run.

- Use SSMS to list all views in a database
 - List columns, triggers, indexes, and statistics
 - Script View As to create scripts for existing views
- Use Transact-SQL:
 - sys.views – lists views in database
 - OBJECT_DEFINITION() – returns the definition of non-encrypted views
 - sys.sql_expression_dependencies – lists objects, including other views, that depend on an object

 **Note:** Despite appearances, each database does not have its own system views. Object Explorer simply gives you a view onto the system views available to all databases.

Catalog Views

There are a number of catalog views that give you information about views, including:

- **sys.objects**
- **sys.views**
- **sys.sql_expression_dependencies**
- **sys.dm_sql_referenced_entities**

The **sys.sql_expression_dependencies** catalog view lets you find column level dependencies. If you change the name of an object that a view references, you must modify the view so that it references the new name. Before renaming an object, it is helpful to display the dependencies of the object so you can determine whether the proposed change will affect any views.

You can find overall dependencies by querying the **sys.sql_expression_dependencies** view. You can find column-level dependencies by querying the **sys.dm_sql_referenced_entities** view.

Display the referenced entities for Person.vwEmployeeList.

Using sys.dm_sql_referenced_entities

```
USE AdventureWorks2016;
GO
SELECT referenced_schema_name, referenced_entity_name, referenced_minor_name,
referenced_class_desc, is_caller_dependent
FROM sys.dm_sql_referenced_entities ('Person.vwEmployeeList', 'OBJECT');
GO
```

For more information about **sys.dm_sql_references_entities**, see Microsoft Docs:

-  [sys.dm_sql_referenced_entities \(Transact-SQL\)](#)
<http://aka.ms/wr0oz6>

For more information about **sys.sql_expression_dependencies**, see Microsoft Docs:

-  [sys.sql_expression_dependencies \(Transact-SQL\)](#)
<http://aka.ms/mgon4g>

System Stored Procedure

You could display object definitions, including unencrypted views, by executing the system stored procedure **sp_helptext** and passing it the name of the view. The view definition will not be displayed, however, if WITH ENCRYPTION was used to create the view.

You can use a system stored procedure to display a view definition.

Display a view definition using sp_helptext

```
USE AdventureWorks2016;
GO
EXEC sp_helptext 'Person.vwEmployeeList';
GO
```

System Function

The **OBJECT_DEFINITION()** function returns the definition of an object in relational format. This is more appropriate for an application to use than the output of a system stored procedure such as **sp_helptext**. Again, the view must not have been created using the WITH ENCRYPTION attribute.

Use a system function to return the view definition.

Return the view definition with OBJECT_DEFINITION()

```
USE AdventureWorks2016;
GO
SELECT OBJECT_DEFINITION (OBJECT_ID(N'Person.vwEmployeeList')) AS [View Definition];
GO
```

Updateable Views

An updateable view lets you modify data in the underlying table or tables. This means that, in addition to being able to query the view, you can also insert, update, or delete rows through the view.

Limitations for Updateable Columns

If you want to update data through your view, you must ensure that the columns:

- Are from one table only.
- Directly reference the base table columns.
- Do not include an aggregate function: AVG, COUNT, SUM, MIN, MAX, GROUPING, STDEV, STDEVP, VAR, and VARP.
- Are not affected by DISTINCT, GROUP BY, HAVING clauses.
- Are not computed columns formed by using any other columns.
- Do not have TOP used in the view definition.

- Data can be modified through a view, providing that:
 - The view includes columns from only one table
 - The columns directly reference the table columns
 - No aggregations or computations
 - The updates comply with the base table constraints
 - NULL or NOT NULL
 - Primary and foreign keys can be enforced
 - WITH CHECK option prevents data being inserted that does not comply with the view definition

Although views can contain aggregated values from the base tables, you cannot update these columns. Columns that are involved in operations, such as GROUP BY, HAVING, or DISTINCT, cannot be updated.

INSTEAD OF Triggers

Updates through views cannot affect columns from more than one base table. However, to work around this restriction, you can create INSTEAD OF triggers. Triggers are discussed in Module 11: *Responding to Data Manipulation via Triggers*.

Updates Must Comply with Table Constraints

Data that is modified through a view must still comply with the base table constraints. This means that updates must still comply with constraints such as NULL or NOT NULL, primary and foreign keys, and defaults. The data must comply with the base table's constraints, as if the base table was being modified directly. This can be difficult if not all the columns in the base table are displayed in the view. For example, an INSERT operation on the view would fail if the base table had NOT NULL columns that were not displayed in the view.

WITH CHECK Option

You can modify a row so that the row no longer meets the view definition. For example, a view might include WHERE State = 'WA'. If a user updated a row and set State = 'CA', this row would not be returned when the view is next queried. It would seem to have disappeared.

If you want to stop updates that do not meet the view definition, specify the WITH CHECK option. SQL Server will then ensure that any data modifications meet the view definition. In the previous example, it would prevent anyone from modifying a row that did not include State = 'WA'.

Hide View Definitions

As we have seen, the view definition can be retrieved from catalog views. As views are sometimes used as a security mechanism to prevent users from seeing data from some columns of a table, developers may want to prevent access to the view definition.

Adding the WITH ENCRYPTION option to a view means that the view definition cannot be retrieved from using `OBJECT_DEFINITION()` or `sp_helptext`. WITH ENCRYPTION also blocks the SCRIPT VIEW AS functionality from SSMS.

If you encrypt your view definitions, it is therefore critical to keep an accurate and up-to-date copy of all view definitions for maintenance purposes.

Whilst encrypting view definitions provides a certain level of security, this also makes it more difficult to troubleshoot when there are performance problems. The encryption is not strong by today's standards—there are third-party tools that can decrypt the source code. Do not rely on this option if protecting the view definition is critical to your business.

Use the WITH ENCRYPTION option to hide the view definition.

WITH ENCRYPTION option

```
CREATE VIEW vwVacationByJobTitle
WITH ENCRYPTION
AS
SELECT JobTitle, Gender, VacationHours
    FROM HumanResources.Employee
    WHERE CurrentFlag = 1;
GO
```

- Use the WITH ENCRYPTION option to obfuscate the view definition
- Limited protection
- Do not rely on WITH ENCRYPTION to protect the view definition

 **Note:** WITH ENCRYPTION also stops the view being published with SQL Server replication.

Demonstration: Creating, Altering, and Dropping a View

In this demonstration, you will see how to:

- Create a view using two AdventureWorksLT tables.
- Query the view and order the result set.
- Alter the view to add the encryption option.
- Drop the view.

Demonstration Steps

1. In SSMS, on the **File** menu, point to **Open**, and then click **File**.
2. In the **Open File** dialog box, navigate to **D:\Demofiles\Mod08**, click **Mod_08_Demo_2A.sql**, and then click **Open**.
3. On the toolbar, in the **Available Databases** list, click **AdventureWorksLT**.

4. Select the code under the **Step 2 - Create a new view** comment, and then click **Execute**.
5. Select the code under the **Step 3 - Query the view** comment, and then click **Execute**.
6. Select the code under the **Step 4 - Query the view and order the results** comment, and then click **Execute**.
7. Select the code under the **Step 5 - Query the view definition via OBJECT_DEFINITION** comment, and then click **Execute**.
8. Select the code under the **Step 6 - Alter the view to use WITH ENCRYPTION** comment, and then click **Execute**.
9. Select the code under the **Step 7 - Requery the view definition via OBJECT_DEFINITION** comment, and then click **Execute**.
10. Note that the query definition is no longer accessible because the view is encrypted.
11. Select the code under the **Step 8 - Drop the view** comment, and then click **Execute**.
12. Close SSMS without saving any changes.

Check Your Knowledge

Question	
What does the WITH CHECK option do?	
Select the correct answer.	
	Checks that the data in the view does not contain mistakes.
	Checks that the view definition is well formed.
	Checks that there is no corruption in the underlying tables.
	Checks that inserted data conforms to the view definition.
	Checks that inserted data complies with table constraints.

Lesson 3

Performance Considerations for Views

This lesson discusses how the query optimizer handles views, what an indexed view is, and when you might use them. It also considers a special type of view—the partitioned view.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how the query optimizer handles views.
- Understand indexed views and when to use them.
- Understand nested views and when to use them.
- Understand partitioned views and when to use them.

Views and Dynamic Resolution

Standard views are expanded and incorporated into the queries in which they are referenced; the objects that they reference are resolved at execution time.

A single query plan is created that merges the query being executed and the definition of any views that it accesses. A separate query plan for the view is not created.

Merging the view query into the outer query is called “inlining” the query. It can be very beneficial to performance because SQL Server can eliminate unnecessary joins and table accesses from queries.

Standard views do not appear in execution plans for queries because the views are not accessed. The underlying objects that they reference will be seen in the execution plans.

You should avoid using `SELECT *` in a view definition. As an example, you will notice that, if you add a new column to the base table, the view will not reflect the column until the view has been refreshed. You can correct this situation by executing an updated `ALTER VIEW` statement or by calling the `sp_refreshview` system stored procedure.

- Dynamic resolution in view optimization can assist performance
- SQL Server will not retrieve data it does not need
- Single query plan for both query and view
- Avoid using `SELECT *` in views

Indexed Views

An indexed view has a clustered index added to it. By adding a clustered index, the view is “materialized” and the data is permanently stored on disk. Complex views that include aggregations and joins can benefit from having an index added to the view. The data stored on disk is faster to retrieve, because any calculations and aggregations do not need to be done at run time.

- An indexed view is materialized and data is stored on disk
- Nonindexed views store only the view definition
- An indexed view is not a table
- Two ways indexed views are used:
 - Directly in a query; in some situations, the indexed view will be faster
 - Indirectly by the query optimizer; when it is beneficial to use the indexed view instead of the underlying tables

Creating Indexed Views

You can create an indexed view using Transact-SQL. Before creating the view, you should check that the SET options on the underlying tables have specific values. The definition of the view must also be deterministic, so that the data stored will always be the same.

To create the view, use the CREATE VIEW <name> WITH SCHEMABINDING statement followed by a CREATE UNIQUE CLUSTERED INDEX statement.

For more information about creating indexed views, see Microsoft Docs:



<http://aka.ms/ypi7xz>

Using Indexed Views

Having created an indexed view, you use it in two ways:

1. It can be referenced directly in a FROM clause. Depending on the query, the indexed view will be faster to access than a nonindexed view. The performance improvement for certain queries can be dramatic when an index is added to a view.
2. The indexed view can be used by the query optimizer, in place of the underlying tables, whenever there is a performance benefit.

When updates to the underlying data are made, SQL Server automatically makes updates to the data that is stored in the indexed view. This means that there is an overhead to using an indexed view, because modifications to data in the underlying tables may not be as quick. Although an indexed view is materialized and the data is stored on disk, it is not a table. The definition of an indexed view is defined by a SELECT statement and the data is modified in the underlying table.

Indexed views have a negative impact on the performance of INSERT, DELETE, and UPDATE operations on the underlying tables because the view must also be updated. However, for some queries, they dramatically improve the performance of SELECT queries on the view. They are most useful for data that is regularly selected, but less frequently updated.



Best Practice: Indexed views are useful in decision support systems that are regularly queried, but updated infrequently. A data warehouse or data mart might use indexed views because much of the data is aggregated for reporting.

Nested View Considerations

A nested view is one that calls another view, which may in turn call one or more other views, and so on. Although the developer of the original view might not have planned it, some views end up as “Russian dolls”, with views inside views, inside views.

SQL Server does not restrict how deep you can nest views, but there are implications to using nesting. Issues that arise include:

- **Broken ownership chains.** To grant permission to others, the view and the underlying table must have the same owner.
- **Performance.** Slow running queries can be difficult to debug when views are nested within one another. In theory, the query optimizer handles the script as if the views were not nested; in practice, it can make bad decisions trying to optimize each part of the code. This type of performance problem can be difficult to debug.
- **Maintenance.** When developers leave, the views they created may still be used in someone else’s code—because the application depends on the view, it cannot be deleted. But no one wants to amend the view because they do not understand the full implications of how the original view is used. The business has to put up with poorly performing queries and views, because it would take too long to go back and understand how the original view is being used.

Having pointed out the pitfalls of nested views, there are also some advantages. After a view has been written, tested, and documented, it can be used in different parts of an application, just like a table. However, it is important to understand the potential problems.

- A nested view is one that calls another view—that view may call another view, and so on
 - Disadvantages include:
 - Broken ownership chains
 - Poorly performing queries that are difficult to debug
 - Problems maintaining tangled code
 - Advantages include:
 - Once a view has been written, tested, and documented, it can be used just like a table

Partitioned Views

A partitioned view is a view onto a partitioned table. The view makes the table appear to be one table, even though it is actually several tables.

Partitioned Tables

To understand partitioned views, we first have to understand partitioned tables. A partitioned table is a large table that has been split into a number of smaller tables. Although the actual size of the table may vary, tables are normally partitioned when performance problems occur, or maintenance jobs take an unacceptable time to complete. To solve these problems, the table is split into a number of smaller tables, using one of the columns as the criteria. For example, a customer table might be partitioned on the date of the last order with a separate table for each year. This speeds up queries, and allows maintenance jobs to complete more quickly. A WITH CHECK constraint is created to ensure that data within each table complies with the constraint. All tables must have the same columns, and all columns must be of the same data type and size.

- A partitioned view is a view onto a partitioned table
 - A partitioned table is a large table that has been divided horizontally
 - All tables have the same columns and data types
 - Use a WITH CHECK constraint
 - Update the underlying tables through the view
 - A partitioned view may be local or distributed
 - Performance benefits
 - Faster querying
 - Faster indexing

In a local partitioned view, all the constituent tables are located on the same SQL Server instance. A distributed partitioned view is where at least one table resides on a different server.

Update Data Using a Partitioned View

A partitioned view is a view onto the constituent tables of a partitioned table. The view allows the tables to be used as if they were one table, so simplifying the management of partitioned tables. You can update the underlying tables through the view; SQL Server ensures that inserts, updates or deletions affect the correct underlying table.

Performance Benefits of Partitioned Views

Large tables benefit from being partitioned because smaller tables are faster to work with. They are faster to query, and faster to index. When used in conjunction with a partitioned view, you can hide the complexity of working with several tables, and simplify application logic.

Question: Can you think of queries in your SQL Server environment that use nested views?

What advantages and disadvantages are there with using nested views?

Lab: Designing and Implementing Views

Scenario

A new web-based stock promotion is being tested at the Adventure Works Bicycle Company. Your manager is worried that providing access from the web-based system directly to the database tables will be insecure, so has asked you to design some views for the web-based system.

The Sales department has also asked you to create a view that enables a temporary worker to enter new customer data without viewing credit card, email address, or phone number information.

Objectives

After completing this lab, you will be able to:

- Create standard views.
- Create updateable views.

Estimated Time: 45 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **AdventureWorks\Student**

Password: **Pa55w.rd**

Exercise 1: Creating Standard Views

Scenario

The web-based stock promotion requires two new views: OnlineProducts and Available Models. The documentation for each view is shown in the following tables:

View 1: OnlineProducts

View Column	Table Column
ProductID	Production.Product,ProductID
Name	Production.Product,Name
Product Number	Production.Product,ProductNumber
Color	Production.Product.Color. If NULL, return 'N/A'
Availability	Production.Product.DaysToManufacture. If 0 returns 'In stock', If 1 returns 'Overnight'. If 2 return '2 to 3 days delivery'. Otherwise, return 'Call us for a quote'.
Size	Production.Product.Size
Unit of Measure	Production.Product.SizeUnitMeasureCode
Price	Production.Product.ListPrice
Weight	Production.Product.Weight

This view is based on the Production.Product table. Products should be displayed only if the product is on sale, which can be determined using the SellStartDate and SellEndDate columns.

View 2: Available Models

View Column	Table Column
Product ID	Production.Product.ProductID
Product Name	Production.Product.Name
Product Model ID	Production.ProductModel.ProductModelID
Product Model	Production.ProductMode.Name

This view is based on two tables: Production.Product and Production.ProductModel. Products should be displayed only if the product is on sale, which can be determined using the SellStartDate and SellEndDate columns.

The main tasks for this exercise are as follows:

1. Prepare the Environment
2. Design and Implement the Views
3. Test the Views

► **Task 1: Prepare the Environment**

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab08\Starter** folder as Administrator.

► **Task 2: Design and Implement the Views**

1. Review the documentation for the new views.
2. Using SSMS, connect to MIA-SQL using Windows Authentication.
3. Open a new query window.
4. Write and execute scripts to create the new views.

► **Task 3: Test the Views**

- Query both views to ensure that they return the data required in the original documentation.

Results: After completing this exercise, you will have two new views in the AdventureWorks database.

Exercise 2: Creating an Updateable View

Scenario

The Sales department has asked you to create an updateable view based on the Sales.CustomerPII table, enabling a temporary worker to enter a batch of new customers while keeping the credit card, email and phone number information secure.

The view must contain three columns from the Sales.CustomerPII table: CustomerID, FirstName and LastName. You must be able to update the view with new customers.

View Columns	Table Columns
CustomerID	Sales.CustomerPII.CustomerID
FirstName	Sales.CustomerPII.FirstName
LastName	Sales.CustomerPII.LastName

The main tasks for this exercise are as follows:

1. Design and Implement the Updateable View
2. Test the Updateable View

► Task 1: Design and Implement the Updateable View

1. Review the requirements for the updateable view.
2. Write and execute a script to create the new view.

► Task 2: Test the Updateable View

1. Write and execute a SELECT query to check that the view returns the correct columns. Order the result set by CustomerID.
2. Write and execute an INSERT statement to add a new record to the view.
3. Check that the new record appears in the view results.
4. Close SSMS without saving any changes.

Results: After completing this exercise, you will have a new updateable view in the database.

Question: What are three requirements for a view to be updateable?

Question: What is a standard, nonindexed view?

Module Review and Takeaways

In this module, we have discussed what a view is, why you would use one, and the advantages of using views. We have looked at system views, and practiced creating views.

You have created views:

- Based on one underlying table.
- Based on two underlying tables.
- Based on one underlying table that can be updated.

We have discussed the problems with nesting views, and the advantages of creating an indexed view.

Review Question(s)

Question: When you create a new view, what does SQL Server store in the database?

Module 9

Designing and Implementing Stored Procedures

Contents:

Module Overview	9-1
Lesson 1: Introduction to Stored Procedures	9-2
Lesson 2: Working with Stored Procedures	9-7
Lesson 3: Implementing Parameterized Stored Procedures	9-16
Lesson 4: Controlling Execution Context	9-21
Lab: Designing and Implementing Stored Procedures	9-25
Module Review and Takeaways	9-29

Module Overview

This module describes the design and implementation of stored procedures.

Objectives

After completing this module, you will be able to:

- Understand what stored procedures are, and what benefits they have.
- Design, create, and alter stored procedures.
- Control the execution context of stored procedures.
- Implement stored procedures that use parameters.

Lesson 1

Introduction to Stored Procedures

Microsoft® SQL Server® database management software includes several built-in system stored procedures, in addition to giving users the ability to create their own. In this lesson, you will learn about the role of stored procedures and the potential benefits of using them. System stored procedures provide a large amount of prebuilt functionality that you can use when you are building applications. Not all Transact-SQL statements are allowed within a stored procedure.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the role of stored procedures.
- Identify the potential benefits of using stored procedures.
- Work with system stored procedures.
- Identify statements that are not permitted within the body of a stored procedure declaration.

What Is a Stored Procedure?

A stored procedure is a named collection of Transact-SQL statements that is stored within the database. Stored procedures offer a way of encapsulating repetitive tasks; they support user-declared variables, conditional execution, and other powerful programming features.

Transact-SQL Code and Logic Reuse

When applications interact with SQL Server, they send commands to the server in one of two ways:

1. The application could send each batch of Transact-SQL commands to the server to be executed, and resend the same commands if the same function needs to be executed again later.
2. Alternatively, a stored procedure could be created at the server level to encapsulate all of the Transact-SQL statements that are required.

Stored procedures are named, and are called by their name. The application can then execute the stored procedure each time it needs that same functionality, rather than sending all of the individual statements that would otherwise be required.

Stored Procedures

Stored procedures are similar to procedures, methods, and functions in high level languages. They can have input and output parameters, in addition to a return value.

A stored procedure can return rows of data; in fact, multiple rowsets can be returned from a single stored procedure.

Stored procedures can be created in either Transact-SQL code or managed .NET code, and are run using the EXECUTE statement.

- When applications interact with SQL Server, there are two basic ways to execute Transact-SQL code
 - Every statement can be issued directly by the application
 - Groups of statements can be stored on the server as stored procedures and given a name—the application then calls the procedures by name
- Stored procedures
 - Are similar to procedures or methods in other languages
 - Can have input parameters
 - Can have output parameters
 - Can return sets of rows
 - Are executed by the EXECUTE Transact-SQL statement
 - Can be created in managed code or Transact-SQL

Benefits of Stored Procedures

Using stored procedures offers several benefits over issuing Transact-SQL code directly from an application.

Security Boundary

Stored procedures can be part of a scheme that helps to increase application security. Users are given permission to execute a stored procedure without being given permission to access the objects that the stored procedure accesses. For example, you can give a user—or set of users via a role—permission to execute a stored procedure that updates a table without granting the user any permissions to the underlying tables.

- Can enhance the security of an application
 - Users can be given permission to execute a stored procedure without permission to the objects that it accesses
- Enables modular programming
 - Create once, but call many times and from many applications
- Enables the delayed binding of objects
 - Can create a stored procedure that references a database object that does not exist yet
- Can improve performance
 - A single statement requested across the network can execute 100s of lines of Transact-SQL code
 - Better opportunities for execution plan reuse

Modular Programming

Code reuse is important. Stored procedures help modular programming by allowing logic to be created once and then reused many times, from many applications. Maintenance is easier because, if a change is required, you often only need to change the procedure, not the application code. Changing a stored procedure could avoid the need to change the data access logic in a group of applications.

Delayed Binding

You can create a stored procedure that accesses (or references) a database object that does not yet exist. This can be helpful in simplifying the order in which database objects need to be created. This is known as deferred name resolution.

Performance

Using stored procedures, rather than many lines of Transact-SQL code, can offer a significant reduction in the level of network traffic.

Transact-SQL code needs to be compiled before it is executed. In many cases, when a stored procedure is compiled, SQL Server will retain and reuse the query plan that it previously generated, avoiding the cost of compiling the code.

Although you can reuse execution plans for ad-hoc Transact-SQL code, SQL Server favors the reuse of stored procedure execution plans. Query plans for ad-hoc Transact-SQL statements are among the first items to be removed from memory when necessary.

The rules that govern the reuse of query plans for ad-hoc Transact-SQL code are largely based on exactly matching the query text. Any difference—for example, white space or casing—will cause a different query plan to be created. The one exception is when the only difference is the equivalent of a parameter.

Overall, however, stored procedures have a much higher chance of achieving query plan reuse.

Working with System Stored Procedures

SQL Server includes a large amount of built-in functionality in the form of system stored procedures and system extended stored procedures.

Types of System Stored Procedure

There are two types of system stored procedure:

1. System stored procedures.
2. System extended stored procedures.

- A large number of system stored procedures are supplied with SQL Server
- Two basic types of system stored procedure
 - **System stored procedures:** typically used for administrative purposes either to configure servers, databases, or objects, or to view information about them
 - **System extended stored procedures:** extend the functionality of SQL Server
- Key difference is how they are coded
 - System stored procedures are Transact-SQL code in the master database
 - System extended stored procedures are references to DLLs

Both these types of system stored procedure are supplied prebuilt within SQL Server. The core difference between the two is that the code for system stored procedures is written in Transact-SQL, and is supplied in the master database that is included in every SQL Server installation. The code for the system extended stored procedures, however, is written in unmanaged native code, typically C++, and supplied via a dynamic-link library (DLL). Since SQL Server 2005, the objects that the procedures access are located in a hidden resource database rather than directly in the master database—but this has no impact on the way they work.

Originally, there was a distinction in the naming of these stored procedures, where system stored procedures had an `sp_` prefix and system extended stored procedures had an `xp_` prefix. Over time, the need to maintain backward compatibility has caused a mixture of these prefixes to appear in both types of procedure. Most system stored procedures still have an `sp_` prefix, and most system extended stored procedures still have an `xp_` prefix, but there are exceptions to both of these rules.

System Stored Procedures

Unlike normal stored procedures, system stored procedures can be executed from within any database without needing to specify the master database as part of their name. Typically, they are used for administrative tasks that relate to configuring servers, databases, and objects or for retrieving information about them. System stored procedures are created within the `sys` schema. Examples of system stored procedures are `sys.sp_configure`, `sys.sp_addmessage`, and `sys.sp_executesql`.

System Extended Stored Procedures

System extended stored procedures are used to extend the functionality of the server in ways that you cannot achieve by using Transact-SQL code alone. Examples of system extended stored procedures are `sys.xp_dirtree`, `sys.xp_cmdshell`, and `sys.sp_trace_create`. (Note how the last example here has an `sp_` prefix.)

User Extended Stored Procedures

Creating user-defined extended stored procedures and attaching them to SQL Server is still possible but the functionality is deprecated, and an alternative should be used where possible.

Extended stored procedures run directly within the memory space of SQL Server—this is not a safe place for users to be executing code. User-defined extended stored procedures are well known to the SQL Server product support group as a source of problems that prove difficult to resolve. Where possible, you should use managed-code stored procedures instead of user-defined extended stored procedures. Creating stored procedures using managed code is covered in Module 13: *Implementing Managed Code in SQL Server*.

Statements Not Permitted in Stored Procedures

Not all Transact-SQL statements can be used within a stored procedure. The following statements are not permitted:

- USE *databasename*
- CREATE AGGREGATE
- CREATE DEFAULT
- CREATE RULE
- CREATE SCHEMA
- CREATE or ALTER FUNCTION
- CREATE or ALTER PROCEDURE
- CREATE or ALTER TRIGGER
- CREATE or ALTER VIEW
- SET PARSEONLY
- SET SHOWPLAN_ALL
- SET SHOWPLAN_TEXT
- SET SHOWPLAN_XML

- Some Transact-SQL statements are not allowed:
 - CREATE AGGREGATE
 - CREATE DEFAULT
 - CREATE or ALTER FUNCTION
 - CREATE or ALTER PROCEDURE
 - SET PARSEONLY
 - SET SHOWPLAN TEXT
 - USE *databasename*
 - CREATE RULE
 - CREATE SCHEMA
 - CREATE or ALTER TRIGGER
 - CREATE or ALTER VIEW
 - SET SHOWPLAN ALL or SET SHOWPLAN XML

Despite these restrictions, it is still possible for a stored procedure to access objects in another database. To access objects in another database, reference them using their three- or four-part name, rather than trying to switch databases with a USE statement.

Demonstration: Working with System Stored Procedures and Extended Stored Procedures

In this demonstration, you will see how to:

- Execute system stored procedures.

Demonstration Steps

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Navigate to the folder **D:\Demofiles\Mod09** and execute **Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. Start SQL Server Management Studio and connect to the **MIA-SQL** instance using Windows authentication.
5. In SQL Server Management Studio, open the file **D:\Demofiles\Mod09\Module09.ssmssln**.
6. In Solution Explorer, in the **Queries** folder, double-click the **11 - Demonstration1A.sql** script file.
7. Highlight the text under the comment **Step 1 - Switch to the AdventureWorks database**, and click **Execute**.

8. Highlight the text under the comment **Step 2 - Execute the sp_configure system stored procedure**, and click **Execute**.
9. Highlight the text under the comment **Step 3 - Execute the xp_dirtree extended system stored procedure**, and click **Execute**.
10. Keep SQL Server Management Studio open for the next demo.

Question: The system stored procedure prefix (sp_) and the extended stored procedure prefix (xp_) have become a little muddled over time. What does this say about the use of prefixes when naming objects like stored procedures?

Lesson 2

Working with Stored Procedures

Now that you know why stored procedures are important, you need to understand the practicalities that are involved in working with them.

Lesson Objectives

After completing this lesson, you will be able to:

- Create a stored procedure.
- Execute stored procedures.
- Use the WITH ROWSETS clause.
- Alter a stored procedure.
- Drop a stored procedure.
- Identify stored procedure dependencies.
- Explain guidelines for creating stored procedures.
- Obfuscate stored procedure definitions.

Creating a Stored Procedure

You use the **CREATE PROCEDURE** Transact-SQL statement to create a stored procedure.

CREATE PROCEDURE is commonly abbreviated to **CREATE PROC**. You cannot replace a procedure by using the **CREATE PROC** statement. In versions of SQL Server before SQL Server 2016 Service Pack 1, you need to alter it explicitly by using an **ALTER PROC** statement or by dropping it and then recreating it. In versions of SQL Server including and after SQL Server 2016 Service Pack 1, you can use the **CREATE OR ALTER** command to create a stored procedure, or alter it if it already exists.

The **CREATE PROC** statement must be the only one in the Transact-SQL batch. All statements from the AS keyword until the end of the script or until the end of the batch (using a batch separator such as GO) will become part of the body of the stored procedure.

Creating a stored procedure requires both the **CREATE PROCEDURE** permission in the current database and the **ALTER** permission on the schema that the procedure is being created in. It is important to keep connection settings such as **QUOTED_IDENTIFIER** and **ANSI_NULLS** consistent when you are working with stored procedures. Stored procedure settings are taken from the settings for the session in which it was created.

Stored procedures are always created in the current database with the single exception of stored procedures that are created with a number sign (#) prefix in their name. The # prefix on a name indicates that it is a temporary object—it is therefore created in the **tempdb** database and removed at the end of the user's session.

- **CREATE PROCEDURE** is used to create new stored procedures
- Before SQL Server 2016 SP1, a procedure must not already exist, otherwise **ALTER** must be used or the procedure dropped first
- In SQL Server 2016 SP1 and later, the **CREATE OR ALTER** statement is supported
- **CREATE PROCEDURE** must be the only statement in a batch

For more information about creating stored procedures, see Microsoft Docs:

 **Create a Stored Procedure**

<https://aka.ms/Hopurd>

Debugging Stored Procedures

When you are working with stored procedures, a good practice is first to write and test the Transact-SQL statements that you want to include in your stored procedure. Then, if you receive the results that you expected, wrap the Transact-SQL statements in a CREATE PROCEDURE statement.



Note: Wrapping the body of a stored procedure with a **BEGIN...END** block is not required but it is considered good practice. Note also that you can terminate the execution of a stored procedure by executing a **RETURN** statement within the stored procedure.

Executing a Stored Procedure

You use the Transact-SQL **EXECUTE** statement to execute stored procedures. **EXECUTE** is commonly abbreviated to **EXEC**.

EXECUTE Statement

The **EXECUTE** statement is most commonly used to execute stored procedures, but can also be used to execute other objects such as dynamic Structured Query Language (SQL) statements.

Using **EXEC**, you can execute system stored procedures within the master database without having to explicitly refer to that database.

Executing user stored procedures in another database requires that you use the three-part naming convention. Executing user stored procedures in a schema other than your default schema requires that you use the two-part naming convention.

- **EXECUTE statement**
 - Used to execute stored procedures and other objects such as dynamic SQL statements stored in a string
 - Use two- or three-part naming when executing stored procedures to avoid SQL Server having to carry out unnecessary searches

Two-Part Naming on Referenced Objects

When you are creating stored procedures, use at least two-part names for referenced objects. If you refer to a table by both its schema name and its table name, you avoid any ambiguity about which table you are referring to, and you maximize the chance of SQL Server being able to reuse query execution plans.

If you use only the name of a table, SQL Server will first search in your default schema for the table. Then, if it does not locate a table that has that name, it will search the dbo schema. This minimizes options for query plan reuse for SQL Server because, until the moment when the stored procedure is executed, SQL Server cannot tell which objects it needs, because different users can have different default schemas.

Two-Part Naming When Creating Stored Procedures

If you create a stored procedure by only supplying the name of the procedure (and not the schema name), SQL Server will attempt to create the stored procedure in your default schema. Scripts that create stored procedures in this way tend to be fragile because the location of the created stored procedure would depend upon the default schema of the user who was executing the script.

Two-Part Naming When Executing Stored Procedures

When you execute a stored procedure, you should supply the name of both the schema and the stored procedure. If you supply only the name of the stored procedure, SQL Server has to attempt to find the stored procedure in several places.

If the stored procedure name starts with sp_ (not recommended for user stored procedures), SQL Server will search locations in the following order, in an attempt to find the stored procedure:

- The sys schema in the master database.
- The default schema for the user who is executing the stored procedure.
- The dbo schema in the current database for the stored procedure.

Having SQL Server perform unnecessary steps to locate a stored procedure reduces performance.

For more information about executing a stored procedure, see Microsoft Docs:



Execute a Stored Procedure

<https://aka.ms/O1i3nv>

Altering a Stored Procedure

The Transact-SQL **ALTER PROCEDURE** statement is used to replace an existing procedure. **ALTER PROCEDURE** is commonly abbreviated to **ALTER PROC**.

ALTER PROC

The main reason for using the **ALTER PROC** statement is to retain any existing permissions on the procedure while it is being changed. Users might have been granted permission to execute the procedure. However, if you drop the procedure, and then recreate it, the permission will be removed and would need to be granted again.

Note that the type of procedure cannot be changed using **ALTER PROC**. For example, a Transact-SQL procedure cannot be changed to a managed-code procedure by using an **ALTER PROCEDURE** statement or vice versa.

- **ALTER PROCEDURE** statement
 - Used to replace a stored procedure
 - Retains the existing permissions on the procedure

Connection Settings

The connection settings, such as **QUOTED_IDENTIFIER** and **ANSI_NULLS**, that will be associated with the modified stored procedure will be those taken from the session that makes the change, not from the original stored procedure—it is important to keep these consistent when you are making changes.

Complete Replacement

Note that, when you alter a stored procedure, you need to resupply any options (such as the **WITH ENCRYPTION** clause) that were supplied while creating the procedure. None of these options are retained and they are replaced by whatever options are supplied in the **ALTER PROC** statement.

Dropping a Stored Procedure

To remove a stored procedure from the database, use the `DROP PROCEDURE` statement, commonly abbreviated to `DROP PROC`.

You can obtain a list of procedures in the current database by querying the **`sys.procedures`** system view.

Dropping a system extended stored procedure is accomplished by executing the stored procedure **`sp_dropextendedproc`**.

Dropping a stored procedure requires either `ALTER` permission on the schema that the procedure is part of, or `CONTROL` permission on the procedure itself.

For more information about deleting a stored procedure, see Microsoft Docs:



Delete a Stored Procedure

<https://aka.ms/Kxgolm>

- **`DROP PROCEDURE`** removes one or more stored procedures from the current database
- `sys.procedures` system view gives details on stored procedures in the current database
- `sp_dropextendedproc` to drop system extended stored procedures

Stored Procedures Error Handling

However well you design a stored procedure, errors can always occur. When you build error handling into your stored procedures, you can be sure that errors will be handled appropriately. The `TRY ... CATCH` construct is the best way to handle errors within a stored procedure.

TRY ... CATCH constructs

Use `TRY ... CATCH` constructs to handle stored procedure errors. This is made up of two blocks of code—a `TRY` block that contains the code for the stored procedure, and a `CATCH` block that contains error handling code. If the code executes without error, the code in the `CATCH` block will not run. However, if the stored procedure does generate an error, then the code in the `CATCH` block can handle the error appropriately.

The `TRY` block starts with the keywords `BEGIN TRY`, and is finished with the keywords `END TRY`. The stored procedure code goes in between `BEGIN TRY` and `END TRY`. Similarly, the `CATCH` block is started with the keywords `BEGIN CATCH`, and is finished with the keywords `END CATCH`.

In this code example, a stored procedure is created to add a new store. It uses the `TRY ... CATCH` construct to catch any errors. In the event of an error, code within the `CATCH` block is executed; in this instance, it returns details about the error.

- Include error handling in your stored procedures
- Use the `TRY ... CATCH` construct to handle errors
- `BEGIN TRY <code> END TRY`
- `BEGIN CATCH <error handling code> END CATCH`
- Error functions used within a `CATCH` block
 - `ERROR_NUMBER()`
 - `ERROR_SEVERITY()`
 - `ERROR_STATE()`
 - `ERROR_PROCEDURE()`
 - `ERROR_LINE()`
 - `ERROR_MESSAGE()`

An example of a stored procedure with error handling code:

The TRY ... CATCH Construct

```

CREATE PROCEDURE Sales.NewStore
    @SalesPersonID AS int,
    @StoreName AS nvarchar(50)

AS
SET NOCOUNT ON;
BEGIN TRY
    INSERT INTO Person.BusinessEntity (rowguid)
        VALUES (DEFAULT);

    DECLARE @BusinessEntityID int = SCOPE_IDENTITY();

    INSERT INTO Sales.Store (BusinessEntityID, [Name], SalesPersonID)
        VALUES (@BusinessEntityID, @StoreName, @SalesPersonID);

END TRY

BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrorNumber,
        ERROR_SEVERITY() AS ErrorSeverity,
        ERROR_STATE() AS ErrorState,
        ERROR_PROCEDURE() AS ErrorProcedure,
        ERROR_LINE() AS ErrorLine,
        ERROR_MESSAGE() AS ErrorMessage;
END CATCH

```

 **Note:** The error functions shown in the example are only used within CATCH blocks. They will return NULL if used outside a CATCH block.

Transaction Handling

You might need to manage transactions within a stored procedure. Perhaps you want to ensure that if one action fails, then all actions are rolled back. Explicit transactions are managed using the BEGIN TRANSACTION and COMMIT TRANSACTION keywords.

In this code example, we create a table with two columns and a primary key constraint. We then create a stored procedure to add rows to the table. We want both inserts to succeed together, or fail together. We do not want one insert to succeed, and the other to fail. To achieve this, we first create a TRY ... CATCH block, as discussed in the previous topic. The two insert statements are enclosed within an explicit transaction. If one fails, both will be rolled back.

- Explicit transactions are managed with
 - BEGIN TRANSACTION or BEGIN TRAN
 - COMMIT TRANSACTION
- Use a TRY ... CATCH block to ROLLBACK transactions
- Use to ensure the complete transaction—or nothing—is committed
- @@TRANCOUNT keeps count of the number of BEGIN TRANSACTIONS
- Use SET XACT_ABORT ON or OFF to determine how SQL Server should handle statements within a transaction

Using transactions within a stored procedure:

Transactions

```
CREATE TABLE MyTable
    (Col1 tinyint PRIMARY KEY, Col2 CHAR(3) NOT NULL);
GO

CREATE PROCEDURE MyStoredProcedure @PriKey tinyint, @CharCol char(3)
AS
BEGIN TRY
    BEGIN TRANSACTION;
    INSERT INTO MyTable VALUES (@PriKey, @CharCol)
    INSERT INTO MyTable VALUES (@PriKey -1, @CharCol)
    COMMIT TRANSACTION;
END TRY

BEGIN CATCH
    ROLLBACK
END CATCH;
GO
```

If you execute the stored procedure with values that enable two rows to be successfully inserted, it will execute successfully, and no transactions will be rolled back. For example:

```
EXEC MyStoredProcedure 1, 'abc'
```

However, if you execute the stored procedure with a value that causes one row to fail, then the catch block will be invoked, and the complete transaction will be rolled back. For example:

```
EXEC MyStoredProcedure 2, 'xyz'
```

In the second instance, the primary key constraint will be violated, and no records will be inserted.

@@TRANCOUNT

The @@TRANCOUNT function keeps count of the number of transactions by incrementing each time a BEGIN TRANSACTION is executed. It decrements by one each time a COMMIT TRANSACTION is executed.

Using the table created in the previous code example, this code fragment shows how @@TRANCOUNT increments and decrements.

@@TRANCOUNT

```
SET XACT_ABORT ON;

SELECT @@TRANCOUNT;
BEGIN TRANSACTION;
SELECT @@TRANCOUNT;
INSERT MyTable VALUES (40, 'abc');
COMMIT TRANSACTION;
SELECT @@TRANCOUNT;
BEGIN TRANSACTION;
INSERT MyTable VALUES (41, 'xyz');
SELECT @@TRANCOUNT;
COMMIT TRANSACTION;
SELECT @@TRANCOUNT

SELECT * FROM MyTable2
```



Note: SET XACT_ABORT ON or OFF determines how SQL Server behaves when a statement fails within a transaction. If SET XACT_ABORT is ON, then the entire transaction is rolled back.

Transactions are discussed in more detail in Module 17.

Stored Procedure Dependencies

Before you drop a stored procedure, you should check for any other objects that are dependent upon the stored procedure. SQL Server includes a number of system stored procedures to help identify stored procedure dependencies.

The **sys.sql_expression_dependencies** view replaces the previous use of the **sp_depends** system stored procedure. The **sys.sql_expression_dependencies** view provides a “one row per name” dependency on user-defined entities in the current database. **sys.dm_sql_referenced_entities** and **sys.dm_sql_referencing_entities** offer more targeted views over the data that the **sys.sql_expression_dependencies** view provides.

You will see an example of how these dependency views are used in the next demonstration.

- New system views replace the use of sp_depends
- **sys.sql_expression_dependencies**
 - Contains one row per dependency by name on user-defined entities in the current database
- **sys.dm_sql_referenced_entities**
 - Contains one row for each entity referenced by another entity
- **sys.dm_sql_referencing_entities**
 - Contains one row for each entity referencing another entity

Guidelines for Creating Stored Procedures

There are several important guidelines that you should consider when you are creating stored procedures.

Qualify Names Inside Stored Procedures

Earlier in this lesson, the importance of using at least a two- or three-part naming convention was discussed. This applies both to the creation of stored procedures and to their execution. If your stored procedure calls another stored procedure, use the fully qualified name.

- Qualify names inside stored procedures
- Keep consistent **SET** options
- SET NOCOUNT ON
- Apply consistent naming conventions (and no **sp_** prefix)
- Use **@@nestlevel** to see current nesting level (32 is the maximum number of levels)
- Use return codes to identify reasons various execution outcomes
- Keep to one procedure for each task

Keeping Consistent SET Options

Database Engine saves the settings of both **SET QUOTED_IDENTIFIER** and **SET ANSI_NULLS** when a Transact-SQL stored procedure is created or altered. These original settings are used when the stored procedure is executed.

SET NOCOUNT ON

Ensure the first statement in your stored procedure is **SET NOCOUNT ON**. This will increase performance by suppressing messages returned to the client following **SELECT**, **INSERT**, **UPDATE**, **MERGE**, and **DELETE** statements.

Applying Consistent Naming Conventions

Avoid creating stored procedures that use sp_ or xp_ as a prefix. SQL Server uses the sp_ prefix to designate system stored procedures and any name that you choose may conflict with a current or future system procedure. An sp_ or xp_ prefix also affects the way SQL Server searches for the stored procedure.

It is important to have a consistent way of naming your stored procedures. There is no right or wrong naming convention but you should decide on a method for naming objects and apply that method consistently. You can enforce naming conventions on most objects by using Policy-Based Management or DDL triggers. These areas are beyond the scope of this course.

Using @@nestlevel to See Current Nesting Level

Stored procedures are nested when one stored procedure calls another or executes managed code by referencing a common language runtime (CLR) routine, type, or aggregate. You can nest stored procedures and managed-code references up to 32 levels. You can use @@nestlevel to check the nesting level of the current stored procedure execution.

Using Return Codes

Return codes are a little like OUTPUT parameters, but return status information about how your stored procedure executed. Return codes are integers that you define to provide relevant information about success, failure, or other possible outcomes during execution. Return codes are flexible, and useful in some circumstances; however, it is good practice to document the meaning of return codes so that they can be used consistently between systems and developers.

Keeping to One Procedure for Each Task

Avoid trying to write one stored procedure to address multiple requirements. Doing so limits the possibilities for reuse and can hinder performance.

Obfuscating Stored Procedures

You can use SQL Server to obfuscate the definition of stored procedures by using the WITH ENCRYPTION clause. However, you must use this with caution because it can make working with the application more difficult and may not achieve the required aims.

The encryption provided by the WITH ENCRYPTION clause is not particularly strong, and is known to be relatively easy to defeat, because the encryption keys are stored in known locations within the encrypted text. There are both direct methods and third-party tools that can reverse the encryption provided by the WITH ENCRYPTION clause.

In addition, encrypted code is much harder to work with in terms of diagnosing and tuning performance issues—so it is important you weigh up the benefits before using the WITH ENCRYPTION clause.

- WITH ENCRYPTION clause
 - Encrypts stored procedure definition stored in SQL Server
 - Protects stored procedure creation logic to a limited extent
 - Is generally not recommended

Demonstration: Stored Procedures

In this demonstration, you will see how to:

- Create, execute, and alter a stored procedure.

Demonstration Steps

1. In Solution Explorer, in the **Queries** folder, double-click the **21 - Demonstration2A.sql** script file.
2. Highlight the code under the comment **Step 1 - Switch to the AdventureWorks database**, and click **Execute**.
3. Highlight the code under the comment **Step 2 - Create the GetBlueProducts stored procedure**, and click **Execute**.
4. Highlight the code under the comment **Step 3 - Execute the GetBlueProducts stored procedure**, and click **Execute**.
5. Highlight the code under the comment **Step 4 - Create the GetBlueProductsAndModels stored procedure**, and click **Execute**.
6. Highlight the code under the comment **Step 5 - Execute the GetBlueProductsAndModels stored procedure which returns multiple rowsets**, and click **Execute**.
7. Highlight the code under the comment **Step 6 - Alter the procedure because the 2nd query does not show only blue products**, and click **Execute**.
8. Highlight the code under the comment **Step 7 - And re-execute the GetBlueProductsAndModels stored procedure**, and click **Execute**.
9. Highlight the code under the comment **Step 8 - Query sys.procedures to see the list of procedures**, and click **Execute**.
10. Keep SQL Server Management Studio open for the next demo.

Check Your Knowledge

Question	
Obfuscating the body of a stored procedure is best avoided, but when might you want to use this functionality?	
Select the correct answer.	
	When transferring the stored procedure between servers.
	When emailing the stored procedure code to a colleague.
	When the stored procedure takes input parameters that should not be disclosed.
	When the stored procedure contains intellectual property that needs protecting.

Lesson 3

Implementing Parameterized Stored Procedures

The stored procedures that you have seen in this module have not involved parameters. They have produced their output without needing any input from the user and they have not returned any values, apart from the rows that they have returned. Stored procedures are more flexible when you include parameters as part of the procedure definition, because you can create more generic application logic. Stored procedures can use both input and output parameters, and return values.

Although the reuse of query execution plans is desirable in general, there are situations where this reuse is detrimental. You will see situations where this can occur and consider options for workarounds to avoid the detrimental outcomes.

Lesson Objectives

After completing this lesson, you will be able to:

- Parameterize stored procedures.
- Use input parameters.
- Use output parameters.
- Explain the issues that surround parameter sniffing and performance, and describe the potential workarounds.

Working with Parameterized Stored Procedures

Parameterized stored procedures enable a much higher level of code reuse. They contain three major components: input parameters, output parameters, and return values.

- Parameterized stored procedures contain three major components
 - Input parameters
 - Output parameters
 - Return values

Input Parameters

Parameters are used to exchange data between stored procedures and the application or tool that called the stored procedure. They enable the caller to pass a data value to the stored procedure. To define a stored procedure that accepts input parameters, you declare one or more variables as parameters in the **CREATE PROCEDURE** statement. You will see an example of this in the next topic.

Output Parameters

Output parameters enable the stored procedure to pass a data value or a cursor variable back to the caller. To use an output parameter within Transact-SQL, you must specify the **OUTPUT** keyword in both the **CREATE PROCEDURE** statement and the **EXECUTE** statement.

Return Values

Every stored procedure returns an integer return code to the caller. If the stored procedure does not explicitly set a value for the return code, the return code is 0 if no error occurs; otherwise, a negative value is returned.

Return values are commonly used to return a status result or an error code from a procedure and are sent by the Transact-SQL **RETURN** statement.

Although you can send a value that is related to business logic via a **RETURN** statement, in general, you should use output parameters to generate values rather than the **RETURN** value.

For more information about using parameters with stored procedures, see Microsoft Docs:



Parameters

<https://aka.ms/Ai6kha>

Using Input Parameters

Stored procedures can accept input parameters in a similar way to how parameters are passed to functions, methods, or subroutines in higher-level languages.

Stored procedure parameters must be prefixed with the @ symbol and must have a data type specified. The data type will be checked when a call is made.

There are two ways to call a stored procedure using input parameters. One is to pass the parameters as a list in the same order as in the CREATE PROCEDURE statement; the other is to pass a parameter name and value pair. You cannot combine these two options in a single EXEC call.

- Parameters have the @ prefix, a data type, and optionally a default value
- Parameters can be passed in order, or by name
- Parameters should be validated early in procedure code

Default Values

You can provide default values for a parameter where appropriate. If a default is defined, a user can execute the stored procedure without specifying a value for that parameter.

An example of a default parameter value for a stored procedure parameter:

An example of a default parameter value for a stored procedure parameter:

Default Values

```
CREATE PROCEDURE Sales.OrdersByDueDateAndStatus
@DueDate datetime,
@Status tinyint = 5
AS
```

Two parameters have been defined (@DueDate and @Status). The @DueDate parameter has no default value and must be supplied when the procedure is executed. The @Status parameter has a default value of 5. If a value for the parameter is not supplied when the stored procedure is executed, a value of 5 will be used.

Validating Input Parameters

As a best practice, validate all incoming parameter values at the beginning of a stored procedure to trap missing and invalid values early. This might include such things as checking whether the parameter is NULL. Validating parameters early avoids doing substantial work in the procedure that then has to be rolled back due to an invalid parameter value.

Executing a Stored Procedure with Input Parameters

Executing a stored procedure that has input parameters is simply a case of providing the parameter values when calling the stored procedure.

This is an example of the previous stored procedure with one input parameter supplied and one parameter using the default value:

This is an example of the previous stored procedure with one input parameter supplied and one parameter using the default value:

Executing a Stored Procedure That Has Input Parameters with a Default Value.

```
EXEC Sales.OrdersByDueDateAndStatus '20050713';
```

This execution supplies a value for both @DueDate and @Status. Note that the names of the parameters have not been mentioned. SQL Server knows which parameter is which by its position in the parameter list.

This is an example of a stored procedure being executed and both parameters are defined by name:

This is an example of a stored procedure being executed and both parameters are defined by name:

Executing a Stored Procedure with Input Parameters

```
EXEC Sales.OrdersByDueDateAndStatus '20050613',8;
```

In this case, the stored procedure is being called by using both parameters, but they are being explicitly identified by name.

In this example, the results will be the same, even though they are in a different order, because the parameters are defined by name:

In this example, the results will be the same, even though they are in a different order, because the parameters are defined by name:

Identifying Parameters by Name

```
EXEC Sales.OrdersByDueDateAndStatus @Status = 5,  
@DueDate = '20050713';
```

Using Output Parameters

Output parameters are declared and used in a similar way to input parameters, but output parameters have a few special requirements.

Requirements for Output Parameters

- You must specify the OUTPUT keyword when you are declaring an output parameter in a stored procedure.
- You must specify the OUTPUT keyword in the list of parameters that are passed with the EXEC statement.

- OUTPUT must be specified
 - When declaring the parameter
 - When executing the stored procedure

The code example shows the declaration of an OUTPUT parameter:

Output Parameter Declaration

```
CREATE PROC Sales.GetOrderCountByDueDate
@dueDate datetime, @OrderCount int OUTPUT
AS
```

In this case, the @DueDate parameter is an input parameter and the @OrderCount parameter has been specified as an output parameter. Note that, in SQL Server, there is no true equivalent of a .NET output parameter. SQL Server OUTPUT parameters are really input/output parameters.

To execute a stored procedure with output parameters you must first declare variables to hold the parameter values. You then execute the stored procedure and retrieve the OUTPUT parameter value by selecting the appropriate variable. The next code example shows this.

This code example shows how to call a stored procedure with input and output parameters:

Stored Procedure with Input and Output Parameters

```
DECLARE @DueDate datetime = '20050713';
DECLARE @OrderCount int;
EXEC Sales.GetOrderCountByDueDate @DueDate, @OrderCount OUTPUT;
SELECT @OrderCount;
```

In the EXEC call, note that the @OrderCount parameter is followed by the OUTPUT keyword. If you do not specify the output parameter in the EXEC statement, the stored procedure would still execute as normal, including preparing a value to return in the output parameter. However, the output parameter value would not be copied back into the @OrderCount variable and you would not be able to retrieve the value. This is a common bug when working with output parameters.

Parameter Sniffing and Performance

When a stored procedure is executed, it is usually good to be able to reuse query plans.

SQL Server attempts to reuse query execution plans from one execution of a stored procedure to the next and, although this is mostly helpful, in some circumstances a stored procedure may benefit from a completely different execution plan for different sets of parameters.

This problem is often referred to as a parameter-sniffing problem, and SQL Server provides a number of ways to deal with it. Note that parameter sniffing only applies to parameters, not to variables within the batch. The code for these looks very similar, but variable values are not "sniffed" at all and this can lead to poor execution plans.

- Query plans generated for a stored procedure are generally reused the next time the stored procedure is executed
 - In most cases this is desirable behavior
- Some stored procedures can benefit from different query plans for different sets of parameters
 - Commonly called a "parameter sniffing" problem
- Options for resolving:
 - **WITH RECOMPILE** in stored procedure code
 - sp_recompile
 - **EXEC WITH RECOMPILE**
 - **OPTION (OPTIMIZE FOR)**

WITH RECOMPILE

You can add a WITH RECOMPILE option when you are declaring a stored procedure. This causes the procedure to be recompiled each time it is executed.

sp_recompile System Stored Procedure

If you call **sp_recompile**, any existing plans for the stored procedure that is passed to it will be marked as invalid and the procedure will be recompiled next time it is executed. You can also pass the name of a table or view to this procedure. In that case, all existing plans that reference the object will be invalidated and recompiled the next time they are executed.

EXEC WITH RECOMPILE

If you add WITH RECOMPILE to the EXEC statement, SQL Server will recompile the procedure before running it and will not store the resulting plan. In this case, the original plan would be preserved and can be reused later.

OPTIMIZE FOR

You use the OPTIMIZE FOR query hint to specify the value of a parameter that should be assumed when compiling the procedure, regardless of the actual value of the parameter.

An example of the OPTIMIZE FOR query hint is shown in the following code example:

OPTIMIZE FOR

```
CREATE PROCEDURE dbo.GetProductNames
@ProductIDLimit int
AS
BEGIN
SELECT ProductID,Name
FROM Production.Product
WHERE ProductID < @ProductIDLimit
OPTION (OPTIMIZE FOR (@ProductIDLimit = 1000))
END;
```

Question: What is the main advantage of creating parameterized stored procedures over nonparameterized stored procedures?

Lesson 4

Controlling Execution Context

Stored procedures normally execute in the security context of the user who is calling the procedure. Providing a chain of ownership extends from the stored procedure to the objects that are referenced, the user can execute the procedure without the need for permissions on the underlying objects. Ownership-chaining issues with stored procedures are identical to those for views. Sometimes you need more precise control over the security context in which the procedure is executing.

Lesson Objectives

After completing this lesson, you will be able to:

- Control execution context.
- Use the EXECUTE AS clause.
- View the execution context.

Controlling Executing Context

The security context in which a stored procedure executes is referred to as its execution context. This context is used to establish the identity against which permissions to execute statements or perform actions are checked.

Execution Contexts

A login token and a user token represent an execution context. The tokens identify the primary and secondary principals against which permissions are checked, and the source that is used to authenticate the token. A login that connects to an instance of SQL Server has one login token and one or more user tokens, depending on the number of databases to which the account has access.

- Security tokens
 - Login token
 - User token
- Control security context using
 - EXECUTE AS

User and Login Security Tokens

A security token for a user or login contains the following:

- One server or database principal as the primary identity.
- One or more principals as secondary identities.
- Zero or more authenticators.
- The privileges and permissions of the primary and secondary identities.

A login token is valid across the instance of SQL Server. It contains the primary and secondary identities against which server-level permissions and any database-level permissions that are associated with these identities are checked. The primary identity is the login itself. The secondary identity includes permissions that are inherited from rules and groups.

A user token is valid only for a specific database. It contains the primary and secondary identities against which database-level permissions are checked. The primary identity is the database user itself. The secondary identity includes permissions that are inherited from database roles. User tokens do not contain server-role memberships and do not honor the server-level permissions that are granted to the identities in the token, including those that are granted to the server-level public role.

Controlling Security Context

Although the default behavior of execution contexts is usually appropriate, there are times when you need to execute within a different security context. This can be achieved by adding the EXECUTE AS clause to the stored procedure definition.

The EXECUTE AS Clause

The EXECUTE AS clause sets the execution context of modules such as stored procedures. It is useful when you need to override the default security context.

Explicit Impersonation

SQL Server supports the ability to impersonate another principal, either explicitly by using the stand-alone EXECUTE AS statement, or implicitly by using the EXECUTE AS clause on modules.

You can use the stand-alone EXECUTE AS statement to impersonate server-level principals, or logins, by using the EXECUTE AS LOGIN statement. You can also use the stand-alone EXECUTE AS statement to impersonate database-level principals, or users, by using the EXECUTE AS USER statement.

To execute as another user, you must first have IMPERSONATE permission on that user. Any login in the sysadmin role has IMPERSONATE permission on all users.

Implicit Impersonation

You can perform implicit impersonations by using the WITH EXECUTE AS clause on modules to impersonate the specified user or login at the database or server level. This impersonation depends on whether the module is a database-level module, such as a stored procedure or function, or a server-level module, such as a server-level trigger.

When you impersonate a principal by using the EXECUTE AS LOGIN statement or within a server-scoped module by using the EXECUTE AS clause, the scope of the impersonation is server-wide. This means that, after the context switch, you can access any resource within the server on which the impersonated login has permissions.

When you impersonate a principal by using the EXECUTE AS USER statement or within a database-scoped module by using the EXECUTE AS clause, the scope of impersonation is restricted to the database. This means that references to objects that are outside the scope of the database will return an error.

- Enables impersonation
- Provides access to modules through impersonation
- Impersonate server-level principals or logins by using **EXECUTE AS LOGIN**
- Impersonate database-level principals or users by using **EXECUTE AS USER**

Viewing Execution Context

Should you need to programmatically query the current security context details, you can use the **sys.login_token** and **sys.user_token** system views to obtain the required information.

sys.login_token System View

The **sys.login_token** system view shows all tokens that are associated with the login. This includes the login itself and the roles of which the user is a member.

- Details of the current security context can be viewed programmatically
- **sys.login_token** shows the login-related details
- **sys.user_token** shows all tokens that are associated with a user

sys.user_token System View

The **sys.user_token** system view shows all tokens that are associated with the user within the database.

Demonstration: Viewing Execution Context

In this demonstration, you will see how to:

- View and change the execution context.

Demonstration Steps

1. In Solution Explorer, expand the **Queries** folder and then double-click the **31 - Demonstration3A.sql** script file.
2. Highlight the code under the comment **Step 1 - Open a new query window to the tempdb database**, and click **Execute**.
3. Highlight the code under the comment **Step 2 - Create a stored procedure that queries sys.login_token and sys.user_token**, and click **Execute**.
4. Highlight the code under the comment **Step 3 - Execute the stored procedure and review the rowsets returned**, and click **Execute**.
5. Highlight the code under the comment **Step 4 - Use the EXECUTE AS statement to change context**, and click **Execute**.
6. Highlight the code under the comment **Step 5 - Try to execute the procedure. Why does it not it work?**, click **Execute** and note the error message.
7. Highlight the code under the comment **Step 6 - Revert to the previous security context**, and click **Execute**.
8. Highlight the code under the comment **Step 7 - Grant permission to SecureUser to execute the procedure**, and click **Execute**.
9. Highlight the code under the comment **Step 8 - Now try again and note the output**, and click **Execute**.
10. Highlight the code under the comment **Step 9 - Alter the procedure to execute as owner**, and click **Execute**.
11. Highlight the code under the comment **Step 10 - Execute as SecureUser again and note the difference**, and click **Execute**.

12. Highlight the code under the comment **Step 11 - Drop the procedure**, and click **Execute**.
13. Close SQL Server Management Studio without saving any changes.

Check Your Knowledge

Question	
What permission is needed to EXECUTE AS another login or user?	
Select the correct answer.	
	sysadmin
	IMPERSONATE
	TAKE OWNERSHIP

Lab: Designing and Implementing Stored Procedures

Scenario

You need to create a set of stored procedures to support a new reporting application. The procedures will be created within a new Reports schema.

Objectives

After completing this lab, you will be able to:

Create a stored procedure.

- Change the execution context of a stored procedure.
- Create a parameterized stored procedure.

Estimated Time: 45 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Create Stored Procedures

Scenario

In this exercise, you will create two stored procedures to support one of the new reports.

Supporting Documentation

Stored Procedure:	Reports.GetProductColors
Input Parameters:	None
Output Parameters:	None
Output Columns:	Color (from Marketing.Product)
Notes:	Colors should not be returned more than once in the output. NULL values should not be returned.
Stored Procedure:	Reports.GetProductsAndModels
Input Parameters:	None
Output Parameters:	None
Output Columns:	ProductID, ProductName, ProductNumber, SellStartDate, SellEndDate and Color (from Marketing.Product), ProductModelID (from Marketing.ProductModel), EnglishDescription, FrenchDescription, ChineseDescription.
Output Order:	ProductID, ProductModelID.
Notes:	For descriptions, return the Description column from the Marketing.ProductDescription table for the appropriate language. The LanguageID for English is "en", for French is "fr" and for Chinese is "zh-cht". If no specific language description is available, return the invariant

Stored Procedure:	Reports.GetProductColors
	language description if it is present. The LanguageID for the invariant language is a blank string ". Where neither the specific language nor invariant language descriptions exist, return the ProductName instead.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Review the Reports.GetProductColors Stored Procedure Specification
3. Design, Create and Test the Reports.GetProductColors Stored Procedure
4. Review the Reports.GetProductsAndModels Stored Procedure Specification
5. Design, Create and Test the Reports.GetProductsAndModels Stored Procedure

► **Task 1: Prepare the Lab Environment**

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
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 wait for the script to finish.

► **Task 2: Review the Reports.GetProductColors Stored Procedure Specification**

- Review the design requirements in the Exercise Scenario for Marketing.GetProductColors.

► **Task 3: Design, Create and Test the Reports.GetProductColors Stored Procedure**

- Design and implement the stored procedure in accordance with the design specifications.

► **Task 4: Review the Reports.GetProductsAndModels Stored Procedure Specification**

- Review the supplied design requirements in the supporting documentation in the Exercise Scenario for Reports.GetProductsAndModels.

► **Task 5: Design, Create and Test the Reports.GetProductsAndModels Stored Procedure**

- Design and implement the stored procedure in accordance with the design specifications.

Results: After completing this lab, you will have created and tested two stored procedures, Reports.GetProductColors and Reports.GetProductsAndModels.

Exercise 2: Create Parameterized Stored Procedures

Scenario

In this exercise, you will create a stored procedure to support one of the new reports.

Supporting Documentation

Stored Procedure	Marketing.GetProductsByColor
Input parameters	@Color (same data type as the Color column in the Production.Product table).
Output parameters	None
Output columns	ProductID, ProductName, ListPrice (returned as a column named Price), Color, Size and SizeUnitMeasureCode (returned as a column named UnitOfMeasure) (from Production.Product).
Output order	ProductName
Notes	The procedure should return products that have no Color if the parameter is NULL.

The main tasks for this exercise are as follows:

1. Review the Reports.GetProductsByColor Stored Procedure Specification
 2. Design, Create and Test the Reports.GetProductsByColor Stored Procedure
- ▶ **Task 1: Review the Reports.GetProductsByColor Stored Procedure Specification**
 - Review the design requirements in the Exercise Scenario for Reports.GetProductsByColor.
 - ▶ **Task 2: Design, Create and Test the Reports.GetProductsByColor Stored Procedure**
 1. Design and implement the stored procedure.
 2. Execute the stored procedure.



Note: Ensure that approximately 26 rows are returned for blue products. Ensure that approximately 248 rows are returned for products that have no color.

Results: After completing this exercise, you will have:

Created the GetProductsByColor parameterized stored procedure.

Exercise 3: Change Stored Procedure Execution Context

Scenario

In this exercise, you will alter the stored procedures to use a different execution context.

The main tasks for this exercise are as follows:

1. Alter the Reports.GetProductColors Stored Procedure to Execute As OWNER
2. Alter the Reports.GetProductsAndModels Stored Procedure to Execute As OWNER.
3. Alter the Reports.GetProductsByColor Stored Procedure to Execute As OWNER

► **Task 1: Alter the Reports.GetProductColors Stored Procedure to Execute As OWNER**

- Alter the stored procedure Reports.GetProductColors so that it executes as owner.

► **Task 2: Alter the Reports.GetProductsAndModels Stored Procedure to Execute As OWNER.**

- Alter the stored procedure Reports.GetProductsAndModels so that it executes as owner.

► **Task 3: Alter the Reports.GetProductsByColor Stored Procedure to Execute As OWNER**

- Alter the stored procedure, Reports.GetProductsByColor so that it executes as owner.

Results: After completing this exercise, you will have altered the three stored procedures created in earlier exercises, so that they run as owner.

Module Review and Takeaways

Best Practice:

- Include the SET NOCOUNT ON statement in your stored procedures immediately after the AS keyword. This improves performance.
- While it is not mandatory to enclose Transact-SQL statements within a BEGIN END block in a stored procedure, it is good practice and can help make stored procedures more readable.
- Reference objects in stored procedures using a two- or three-part naming convention. This reduces the processing that the database engine needs to perform.

Avoid using SELECT * within a stored procedure even if you need all columns from a table. Specifying the column names explicitly reduces the chance of issues, should columns be added to a source table.

Module 10

Designing and Implementing User-Defined Functions

Contents:

Module Overview	10-1
Lesson 1: Overview of Functions	10-2
Lesson 2: Designing and Implementing Scalar Functions	10-5
Lesson 3: Designing and Implementing Table-Valued Functions	10-10
Lesson 4: Considerations for Implementing Functions	10-14
Lesson 5: Alternatives to Functions	10-20
Lab: Designing and Implementing User-Defined Functions	10-22
Module Review and Takeaways	10-24

Module Overview

Functions are routines that you use to encapsulate frequently performed logic. Rather than having to repeat the function logic in many places, code can call the function. This makes code more maintainable, and easier to debug.

In this module, you will learn to design and implement user-defined functions (UDFs) that enforce business rules or data consistency. You will also learn how to modify and maintain existing functions.

Objectives

After completing this module, you will be able to:

- Describe different types of functions.
- Design and implement scalar functions.
- Design and implement table-valued functions (TVFs).
- Describe considerations for implementing functions.
- Describe alternatives to functions.

Lesson 1

Overview of Functions

Functions are routines that consist of one or more Transact-SQL statements that you can use to encapsulate code for reuse. A function takes zero or more input parameters and returns either a scalar value or a table. Functions do not support output parameters but do return results, either as a single value or as a table.

This lesson provides an overview of UDFs and system functions.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe different types of functions.
- Use system functions.

Types of Functions

Most high level programming languages offer functions as blocks of code that are called by name and can process input parameters. Microsoft® SQL Server® offers three types of functions: scalar functions, TVFs, and system functions.

You can create two types of TVFs: inline TVFs and multistatement TVFs.

 **Note:** Functions cannot modify data or insert rows into a database.

- Types of functions:
 - Scalar functions
 - Table-valued functions
 - Inline versus multistatement functions
 - System functions
 - Functions cannot modify data

Scalar Functions

Scalar functions return a single data value of the type that is defined in a RETURNS clause. An example of a scalar function would be one that extracts the protocol from a URL. From the string "http://www.microsoft.com", the function would return the string "http".

Inline Table-Valued Functions

An inline TVF returns a table that is the result of a single SELECT statement. This is similar to a view, but an inline TVF is more flexible, in that parameters can be passed to the SELECT statement within the function.

For example, if a table holds details of sales for an entire country, you could create individual views to return details of sales for particular states. You could write an inline TVF that takes the state code or ID as a parameter, and returns sales data for the state that match the parameter. In this way, you would only need a single function to provide details for all states, rather than separate views for each state.

Multistatement Table-Valued Functions

A multistatement TVF returns a table derived from one or more Transact-SQL statements. It is similar to a stored procedure. Multistatement TVFs are created for the same reasons as inline TVFs, but are used when the logic that the function needs to implement is too complex to be expressed in a single SELECT statement. You can call them from within a FROM clause.

System Functions

System functions are provided with SQL Server to perform a variety of operations. You cannot modify them. System functions are described in the next topic.

For more details about the restrictions and usage of UDFs, see Microsoft Docs:



Create User-Defined Functions (Database Engine)

<http://aka.ms/fuhvvs>

System Functions

SQL Server has a wide variety of system functions that you can use in queries to return data or to perform operations on data. System functions are also known as built-in functions.

 **Note:** Virtual functions provided in earlier versions of SQL Server had names with a **fn_** prefix. These are now located in the sys schema.

- SQL Server includes a large number of built-in functions
- Rowset functions: for example, OPENQUERY, OPENROWSET
- Aggregate functions: for example, AVG, MAX, SUM
- Ranking functions: for example, RANK, ROW_NUMBER
- Scalar functions include:
 - Configuration functions
 - Conversion functions
 - Cursor functions
 - Date and time functions
 - Mathematical functions
 - Security functions

Scalar Functions

Most system functions are scalar functions. They provide the functionality that is commonly provided by functions in other high level languages, such as operations on data types (including strings and dates and times) and conversions between data types. SQL Server provides a library of mathematical and cryptographic functions. Other functions provide details of the configuration of the system, and its security.

Rowset Functions

These return objects that can be used instead of Transact-SQL table reference statements. For example, OPENJSON is a SQL Server function that can be used to import JSON into SQL Server or transform JSON to a relational format.

Aggregate Functions

Aggregates such as MIN, MAX, AVG, SUM, and COUNT perform calculations across groups of rows. Many of these functions automatically ignore NULL rows.

Ranking Functions

Functions such as ROW_NUMBER, RANK, DENSE RANK, and NTILE perform windowing operations on rows of data.

For more information about system functions that return values, settings and objects, see Microsoft Docs:



System Functions (Transact-SQL)

<http://aka.ms/ouqysl>

For more information about different types of system functions, see Microsoft Docs:



What are the SQL database functions?

<http://aka.ms/jw8w5j>

Apart From Data Type and Data Time, Which SQL Server Functions Have You Used?

Responses will vary, based on experience.

Check Your Knowledge

Question	
Which of these is a ranking function?	
Select the correct answer.	
	OPENROWSET
	ROWCOUNT_BIG
	GROUPING_ID
	ROW_NUMBER
	OPENXML

Lesson 2

Designing and Implementing Scalar Functions

You have seen that functions are routines that consist of one or more Transact-SQL statements that you can use to encapsulate code for reuse—and that functions can take zero or more input parameters, and return either scalar values or tables.

This lesson provides an overview of scalar functions and explains why and how you use them, in addition to explaining the syntax for creating them.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe a scalar function.
- Create scalar functions.
- Explain deterministic and nondeterministic functions.

What Is a Scalar Function?

You use scalar functions to return information from a database. A scalar function returns a single data value of the type that is defined in a RETURN clause.

Scalar Functions

Scalar functions are created using the CREATE FUNCTION statement. The body of a function is defined within a BEGIN...END block. The function body contains the series of Transact-SQL statements that return the value.

Consider the function definition in the following code example:

CREATE FUNCTION

```
CREATE FUNCTION dbo.ExtractProtocolFromURL
( @URL nvarchar(1000))
RETURNS nvarchar(1000)
AS BEGIN
RETURN CASE WHEN CHARINDEX(N':',@URL,1) >= 1
THEN SUBSTRING(@URL,1,CHARINDEX(N':',@URL,1) - 1)
END;
END;
```

Scalar functions:

- Return a single data value
- Can return any data type except **rowversion**, **cursor**, and **table** when implemented in Transact-SQL
- Can return any data type except for **rowversion**, **cursor**, **table**, **text**, **ntext**, and **image** when implemented in managed code



Note: Note that the body of the function consists of a single RETURN statement that is wrapped in a BEGIN...END block.

You can use the function in the following code example as an expression, wherever a single value could be used:

Using a Function As an Expression

```
SELECT dbo.ExtractProtocolFromURL(N'http://www.microsoft.com');
IF (dbo.ExtractProtocolFromURL(@URL) = N'http')
...
...
```

You can also implement scalar functions in managed code. Managed code will be discussed in Module 13: *Implementing Managed Code in SQL Server*. The allowable return values for scalar functions differ between functions that are defined in Transact-SQL and functions that are defined by using managed code.

Creating Scalar Functions

User-defined functions are created by using the **CREATE FUNCTION** statement, modified by using the **ALTER FUNCTION** statement, and removed by using the **DROP FUNCTION** statement. Even though you must wrap the body of the function (apart from inline functions) in a BEGIN ... END block, **CREATE FUNCTION** must be the only statement in the batch.

In SQL Server 2016 Service Pack 1 and later, you can use the **CREATE OR ALTER** statement to add a function, or update its definition if the function already exists.

- Scalar UDFs:
 - Return a single data type from a database
 - Usually include parameters
 - Use two-part naming
 - Stop on error
 - Cannot have side effects
 - CREATE FUNCTION must be the only statement in a batch

 **Note:** Altering a function retains any permissions already associated with it.

Scalar User-Defined Functions (UDFs)

You use scalar functions to return information from a database. A scalar function returns a single data value of the type defined in a RETURNS clause. The body of the function, which is defined in a BEGIN...END block, contains the series of Transact-SQL statements that return the value.

Guidelines

Consider the following guidelines when you create scalar UDFs:

- Make sure that you use two-part naming for the function and for all database objects that the function references.
- Avoid Transact-SQL errors that lead to a statement being canceled and the process continuing with the next statement in the module (such as within triggers or stored procedures) because they are treated differently inside a function. In functions, such errors cause the execution of the function to stop.

Side Effects

A function that modifies the underlying database is considered to have side effects. In SQL Server, functions are not permitted to have side effects. You cannot change data in a database within a function; you should not call a stored procedure; and you cannot execute dynamic Structured Query Language (SQL) code.

For more information about the create function, see Microsoft Docs:

Create Function (Transact-SQL)

<http://aka.ms/jh54s1>

This is an example of a CREATE FUNCTION statement. This function calculates the area of a rectangle, when four coordinates are entered. Note the use of the ABS function, an in-built mathematical function that returns a positive value (absolute value) for a given input.

Example of the CREATE FUNCTION statement to create a scalar UDF:

CREATE FUNCTION

```
CREATE FUNCTION dbo.RectangleArea
(@X1 float, @Y1 float, @X2 float, @Y2 float)
RETURNS float
AS BEGIN
RETURN ABS(@X1 - @X2) * ABS(@Y1 - @Y2);
END;
```

Deterministic and Nondeterministic Functions

Both built-in functions and UDFs fall into one of two categories: deterministic and nondeterministic. This distinction is important because it determines where you can use a function. For example, you cannot use a nondeterministic function in the definition of a calculated column.

Deterministic Functions

A deterministic function is one that will always return the same result when it is provided with the same set of input values for the same database state.

Consider the function definition in the following code example:

Deterministic Function

```
CREATE FUNCTION dbo.AddInteger
(@FirstValue int, @SecondValue int)
RETURNS int
AS BEGIN
RETURN @FirstValue + @SecondValue;
END;
GO
```

- Deterministic functions
 - Always return the same result given the same input (and the same database state)
- Nondeterministic
 - May return different results given a specific input
- Built-in functions
 - Can be deterministic or nondeterministic

Every time the function is called with the same two integer values, it will return exactly the same result.

Nondeterministic Functions

A nondeterministic function is one that may return different results for the same set of input values each time it is called, even if the database remains in the same state. Date and time functions are examples of nondeterministic functions.

Consider the function in the following code example:

Nondeterministic Function

```
CREATE FUNCTION dbo.CurrentUTCTimeAsString()
RETURNS varchar(40)
AS BEGIN
RETURN CONVERT(varchar(40),SYSUTCDATETIME(),100);
END;
```

Each time the function is called, it will return a different value, even though no input parameters are supplied.

You can use the **OBJECTPROPERTY()** function to determine if a UDF is deterministic.

For more information about deterministic and nondeterministic functions, see Microsoft Docs:

Deterministic and Nondeterministic Functions

<http://aka.ms/doi8in>

The following code example creates a function, and then uses OBJECTPROPERTY to return whether or not it is deterministic. Note the use of the OBJECT_ID function to return the ID of the TodayAsString function.

Using OBJECTPROPERTY to show whether or not a function is deterministic:

OBJECTPROPERTY

```
CREATE FUNCTION dbo.TodayAsStringC(@Format int= 112)
RETURNS VARCHAR (20)
AS BEGIN
RETURN CONVERT (VARCHAR (20),
CAST(SYSDATETIME() AS date), @Format);
END;
GO

SELECT OBJECTPROPERTY(OBJECT_ID('dob.TodayAsString'), 'IsDeterministic');
```

Demonstration: Working with Scalar Functions

In this demonstration, you will see how to:

- Work with scalar functions.

Demonstration Steps

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **D:\Demofiles\Mod10\Setup.cmd** as an administrator.
3. In the **User Access Control** dialog box, click **Yes**.
4. Wait for **setup.cmd** to complete successfully.
5. On the taskbar, click **Microsoft SQL Server Management Studio**.
6. In the **Connect to Server** dialog box, in **Server** name, type **MIA-SQL** and then click **Connect**.
7. On the **File** menu, point to **Open**, and then click **Project/Solution**.
8. In the **Open Project** dialog box, navigate to **D:\Demofiles\Mod10\Demo10.ssmssqlproj**, and then click **Open**.

9. In Solution Explorer, expand the **Queries** folder, and then double-click **21 - Demonstration 2A.sql**.
10. Select the code under **Step A** to use the **tempdb** database, and then click **Execute**.
11. Select the code under **Step B** to create a function that calculates the end date of the previous month, and then click **Execute**.
12. Select the code under **Step C** to query the function, and then click **Execute**.
13. Select the code under **Step D** to establish if the function is deterministic, and then click **Execute**.
14. Select the code under **Step E** to drop the function, and then click **Execute**.
15. Create a function under **Step F** using the EOMONTH function (the code should resemble the following), and then click **Execute**.

```
CREATE FUNCTION dbo.EndOfPreviousMonth (@DateToTest date)
RETURNS date
AS BEGIN
    RETURN EOMONTH ( dateadd(month, -1, @DateToTest ) );
END;
GO
```

16. Select the code under **Step G** to query the new function, and then click **Execute**.
17. Select the code under **Step H** to drop the function, and then click **Execute**.
18. Keep SQL Server Management Studio open with the **Demo10.ssmssqlpro** solution loaded for the next demo.

Check Your Knowledge

Question
Which of these data types can be returned by a scalar function used in managed code?
Select the correct answer.
rowversion
table
cursor
integer

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? A deterministic function is one that may return different results for the same set of input values each time it is called, even if the database remains in the same state.	

Lesson 3

Designing and Implementing Table-Valued Functions

In this lesson, you will learn how to work with functions that return tables instead of single values. These functions are known as table-valued functions (TVFs). There are two types of TVFs: inline and multistatement.

The ability to return a table of data is important because it means a function can be used as a source of rows in place of a table in a Transact-SQL statement. In many cases, this avoids the need to create temporary tables.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe TVFs.
- Describe inline TVFs.
- Describe multistatement TVFs.

What Are Table-Valued Functions?

Unlike scalar functions, TVFs return a table that can contain many rows of data, each with many columns.

Table-Valued Functions

There are two types of TVFs you can deploy.

Inline TVFs

These return an output table that is defined by a **RETURN** statement that consists of a single **SELECT** statement.

- Table-valued functions
 - TVFs return a **TABLE** data type
 - Inline TVFs have a function body with only a single **SELECT** statement
 - Multistatement TVFs construct, populate, and return a table within the function
 - TVFs are queried like a table
 - TVFs are often used like parameterized views

Multistatement TVFs

If the logic of the function is too complex to include in a single **SELECT** statement, you need to implement the function as a multistatement TVF. Multistatement TVFs construct a table within the body of the function, and then return the table. They also need to define the schema of the table to be returned.



Note: You can use both types of TVF as the equivalent of parameterized views.

For more information about TVFs, see Microsoft TechNet:



Table-Valued User-Defined Functions

<http://aka.ms/hx8dvv>

Inline Table-Valued Functions

You can use inline functions to achieve the functionality of parameterized views. One of the limitations of a view is that you cannot include a user-provided parameter within the view when you create it.

In the following code example, note that the return type is **TABLE**. The definition of the columns of the table is not shown. You do not explicitly define the schema of the returned table. The output table schema is derived from the **SELECT** statement that you provide within the **RETURN** statement. Every column that the **SELECT** statement returns should also have a distinct name.

Example of a table-valued function:

Table valued function

```
USE AdventureWorks;
GO

CREATE FUNCTION Sales.GetLastOrdersForCustomer
(@CustomerID int, @NumberOfOrders int)
RETURNS TABLE
AS
RETURN (SELECT TOP (@NumberOfOrders) soh.SalesOrderID, soh.OrderDate,
soh.PurchaseOrderNumber
    FROM Sales.SalesOrderHeader AS soh
    WHERE soh.CustomerID = @CustomerID
    ORDER BY soh.OrderDate DESC
);

GO

SELECT * FROM Sales.GetLastOrdersForCustomer (17288,2)
```

- Inline table-valued functions
- Returns a single result set
- There is no function body with BEGIN and END
- The returned table definition taken is from the SELECT statement
- Can be seen as a parameterized view

For inline functions, the body of the function is not enclosed in a BEGIN...END block. A syntax error occurs if you attempt to use this block. The **CREATE FUNCTION** statement—or **CREATE OR ALTER** statement, in versions where it is supported—still needs to be the only statement in the batch.

For more information about inline TVFs, see Microsoft Technet:

Inline User-Defined Functions

<http://aka.ms/Abkksx>

Multistatement Table-Valued Functions

You use a multistatement TVF to create more complex queries that determine which rows are returned in the table. You can use UDFs that return a table to replace views. This is very useful when the logic that is required for constructing the return table is more complex than would be possible within the definition of a view.

A TVF, like a stored procedure, can use complex logic and multiple Transact-SQL statements to build a table.

In the example on the slide, a function is created that returns a table of dates. For each row, two columns are returned: the position of the date within the range of dates, and the calculated date. The system does not already include a table of dates, so a loop needs to be constructed to calculate the required range of dates. You cannot implement this in a single **SELECT** statement unless another object, such as a table of numbers, is already present in the database. In each iteration of the loop, an **INSERT** operation is performed on the table that is later returned.

In the same way that you use a view, you can use a TVF in the FROM clause of a Transact-SQL statement.

The following code is an example of a multistatement TVF:

Multistatement TVF

```
CREATE FUNCTION dbo.GetDateRange (@StartDate date, @NumberOfDays int)
    RETURNS @DateList TABLE
        (Position int, DateValue date)
AS BEGIN
    DECLARE @Counter int = 0;
    WHILE (@Counter < @NumberOfDays) BEGIN
        INSERT INTO @DateList
        VALUES (@Counter + 1, DATEADD (day, @Counter, @StartDate));
        SET @Counter += 1;
    END;
    RETURN;
END;
GO

SELECT* FROM dbo.GetDateRange('2009-12-31', 14);
```

- Function body is enclosed by BEGIN and END
- Definition of returned table must be supplied
- Table variable is populated within function body and then returned

Demonstration: Implementing Table-Valued Functions

In this demonstration, you will see how to:

- Implement TVFs.

Demonstration Steps

1. In Solution Explorer, expand the **Queries** folder, and then double-click **31 - Demonstration 3A.sql**.
2. Select the code under **Step A** to use the AdventureWorks database, and then click **Execute**.
3. Select the code under **Step B** to create a table-valued function, and then click **Execute**.
4. Select the code under **Step C** to query the function, and then click **Execute**.

5. Select the code under **Step D** to use CROSS APPLY to call the function, and then click **Execute**.
6. Select the code under **Step E** to drop the function, and then click **Execute**.
7. Close SQL Server Management Studio without saving any changes.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? Both scalar functions and TVFs can return a table that might contain many rows of data, each with many columns—TVFs are a simpler way of returning data in tables.	

Question: You have learned that TVFs return tables. What are the two types of TVF and how do they differ?

Lesson 4

Considerations for Implementing Functions

Although it's important to create functions in Transact-SQL, there are some considerations you need to be aware of. For example, you should avoid negative performance impacts through inappropriate use of functions—a common issue. This lesson provides guidelines for the implementation of functions, and describes how to control their security context.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the performance impacts of scalar functions.
- Describe the performance impacts of table-valued functions.
- Control the execution context.
- Use the EXECUTE AS clause.
- Explain some guidelines for creating functions.

Performance Impacts of Scalar Functions

The code for views is incorporated directly into the code for the query that accesses the view. This is not the case for scalar functions.

Common Performance Problems

The overuse of scalar functions is a common cause of performance problems in SQL Server systems. For example, a WHERE clause predicate that calls a scalar function for every target row. In many cases, extracting the code from the function definition and incorporating it directly into the query will resolve the performance issue. You will see an example of this in the next lab.

Example of code that will not perform well:

Using a Scalar Function Within a SELECT Statement

```
SELECT soh.CustomerID, soh.SalesOrderID
    FROM Sales.SalesorderHeader AS soh
    WHERE soh.SalesOrderID = dbo.GetLatestBulkOrder(soh.CustomerID)
ORDER BY soh.CustomerID, soh.SalesOrderID;
```

- The code for scalar functions is not incorporated into the query
- Different to views where the code is incorporated into the query
- Scalar functions used in SELECT lists or WHERE clause predicates can impact performance

Performance Impacts of Table-Valued Functions

The code for a TVF may or may not be incorporated into the query that uses the function, depending on the type of TVF. Inline TVFs are directly incorporated into the code of the query that uses them.

Common Performance Problems

Multistatement TVFs are not incorporated into the code of the query that uses them. The inappropriate usage of such TVFs is a common cause of performance issues in SQL Server.

- The code for inline TVFs is incorporated into the surrounding query
- The code for multistatement TVFs is not incorporated into the surrounding query
- Performance can be poor except where executed only once in a query
- Very common cause of performance problems
- CROSS APPLY can cause TVFs to be repeatedly executed

You can use the **CROSS APPLY** operator to call a TVF for each row in the table on the left within the query. Designs that require the calling of a TVF for every row in a table can lead to significant performance overhead. You should examine the design to see if there is a way to avoid the need to call the function for each row.

Another example of poorly performing code:

Calling a Function for Every Row

```
SELECT c.customerID, c.AccountNumber, glofc.SalesOrderID, glofc.OrderDate
  FROM sales.Customer AS c
  CROSS APPLY
    sales.GetLastordersForCustomer(c.customerID, 3) AS glofc
 ORDER BY c.CustomerID, glofc.SalesOrderID;
```

 **Note:** Interleaved execution, introduced as part of the Adaptive Query Processing feature of SQL Server 2017, is designed to improve the performance of multistatement TVFs by generating more accurate cardinality estimates, leading to more accurate query execution plans. Adaptive Query Processing is enabled for all databases with a compatibility level of 140 or higher.

Controlling the Execution Context

Execution context establishes the identity against which permissions are checked. The user or login that is connected to the session, or calling a module (such as a stored procedure or function), determines the execution context.

When you use the EXECUTE AS clause to change the execution context so that a code module executes as a user other than the caller, the code is said to "impersonate" the alternative user.

Before you can create a function that executes as another user, you need to have IMPERSONATE permission on that user, or be part of the **dbo** role.

- If a function and a table have the same owner, users can be given access to data via the function even if they do not have access to the table
 - Enables filtered access to tables
- If a function and a table have different owners, a user will not have access to data via the function, even if the owner of the function has access to the table

The EXECUTE AS Clause

The **EXECUTE AS** clause sets the execution context of a session.

You can use the **EXECUTE AS** clause in a stored procedure or function to set the identity that is used as the execution context for the stored procedure or function.

EXECUTE AS means you can create procedures that execute code that the user who is executing the procedure is not permitted to execute. In this way, you do not need to be concerned about broken ownership chains or dynamic SQL execution.

SQL Server supports the ability to impersonate another principal, either explicitly by using the stand-alone **EXECUTE AS** statement, or implicitly by using the **EXECUTE AS** clause on modules. You can use the stand-alone **EXECUTE AS** statement to impersonate server-level principals, or logins, by using the **EXECUTE AS LOGIN** statement. You can also use the stand-alone **EXECUTE AS** statement to impersonate database-level principals, or users, by using the **EXECUTE AS USER** statement.

Implicit impersonations that are performed through the **EXECUTE AS** clause on modules impersonate the specified user or login at the database or server level. This impersonation depends on whether the module is a database-level module, such as a stored procedure or function, or a server-level module, such as a server-level trigger.

When you are impersonating a principal by using the **EXECUTE AS LOGIN** statement, or within a server-scoped module by using the **EXECUTE AS** clause, the scope of the impersonation is server-wide. This means that, after the context switch, you can any resource within the server on which the impersonated login has permissions.

However, when you are impersonating a principal by using the **EXECUTE AS USER** statement, or within a database-scoped module by using the **EXECUTE AS** clause, the scope of impersonation is restricted to the database by default. This means that references to objects that are outside the scope of the database will return an error.

Execute As:

Syntax for Execute As

```
CREATE FUNCTION Sales.GetOrders
    RETURNS TABLE
    WITH EXECUTE AS {CALLER SELF 1 OWNER 'user_name'}
    AS
    ....
```

- The **EXECUTE AS** clause:
 - Enables impersonation
 - Provides access to modules via impersonation
 - Can be used to impersonate server-level principals or logins via the **EXECUTE AS LOGIN** statement
 - Can be used to impersonate database level principals or users via the **EXECUTE AS USER** statement

Guidelines for Creating Functions

Consider the following guidelines when you create user-defined functions (UDFs):

- **UDF Performance.** In many cases, inline functions are much faster than multistatement functions. Wherever possible, try to implement functions as inline functions.
- **UDF Size.** Avoid building large, general-purpose functions. Keep functions relatively small and targeted at a specific purpose. This will avoid code complexity, but will also increase the opportunities for reusing the functions.
- **UDF Naming.** Use two-part naming to qualify the name of any database objects that are referred to within the function. You should also use two-part naming when you are choosing the name of the function.
- **UDFs and Exception Handling.** Avoid statements that will raise Transact-SQL errors, because exception handling is not permitted within functions.
- **UDFs with Indexes.** Consider the impact of using functions in combination with indexes. In particular, note that a WHERE clause that uses a predicate, such as the following code example, is likely to remove the usefulness of an index on **CustomerID**:

- Determine function type
- Create one function for one task
- Qualify object names inside function
- Consider the performance impacts of how you intend to use the function
 - Particular issues exist with the inability to index function results
- Functions cannot contain structured exception handling

For example, consider the function definition in the following code fragment:

Functions with Indexes

```
WHERE Function(CustomerID) = Value
```

Demonstration: Controlling the Execution Context

In this demonstration, you will see how to:

- Alter the execution context of a function.

Demonstration Steps

Alter the Execution Context of a Function

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **D:\Demofiles\Mod10\Setup.cmd** as an administrator.
3. In the **User Access Control** dialog box, click **Yes**.
4. On the taskbar, click **Microsoft SQL Server Management Studio**.
5. In the **Connect to Server** dialog box, in **Server name** box, type **MIA-SQL**, and then click **Connect**.
6. On the **File** menu, point to **Open**, click **Project/Solution**.
7. In the **Open Project** dialog box, navigate to **D:\Demofiles\Mod10\Demo10.ssmssqlproj**, and then click **Open**.

8. In Solution Explorer, expand the **Queries** folder, and then double-click **41 - Demonstration 4A.sql**.
9. Select the code under **Step A** to use the master database, and then click **Execute**.
10. Select the code under **Step B** to create a test login, and then click **Execute**.
11. Select the code under **Step C** to use the **AdventureWorks** database and create a user, and then click **Execute**.
12. Select the code under **Step D** to create a function with default execution context, and then click **Execute**.
13. Select the code under **Step E** to try to add WITH EXECUTE AS, and then click **Execute**.
14. Select the code under **Step F** to recreate the function as a multistatement table-valued function, and then click **Execute**.
15. Select the code under **Step G** to select from the function, and then click **Execute**.
16. Select the code under **Step H** to drop the objects, and then click **Execute**.
17. Close SQL Server Management Studio without saving any changes.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? The underuse of scalar functions is a common cause of performance problems in SQL Server systems.	

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? Before you can create a function that executes as another user, you need to have IMPERSONATE permission on that user, or be part of the dbo role.	

Check Your Knowledge

Question
Which of these is not considered when creating UDFs?
Select the correct answer.
Function Naming
Function Code Size
Function Log Size
Function Exception Handling
Function Performance

Lesson 5

Alternatives to Functions

Functions are one option for implementing code. This lesson explores situations where other solutions may be appropriate and helps you choose which solution to use.

Lesson Objectives

After completing this lesson, you will be able to:

- Compare table-valued functions and stored procedures.
- Compare table-valued functions and views.

Comparing Table-Valued Functions and Stored Procedures

You can often use TVFs and stored procedures to achieve similar outcomes. However, not all client applications can call both. This means that you cannot necessarily use them interchangeably. Each approach also has its pros and cons.

Although you can access the output rows of a stored procedure by using an **INSERT EXEC** statement, it is easier to consume the output of a function in code than the output of a stored procedure.

For example, you cannot execute the following code:

Cannot Select from a Stored Procedure

```
SELECT * FROM (EXEC dbo.GetCriticalPathNodes);
```

You could assign the output of a function to a variable in code.

Stored procedures can modify data in database tables. Functions cannot modify data in database tables. Functions that include such "side effects" are not permitted. Functions can have significant performance impacts when they are called for each row in a query; for example, when a TVF is called by using a **CROSS APPLY** or **OUTER APPLY** statement.

Stored procedures can execute dynamic SQL statements. Functions are not permitted to execute dynamic SQL statements.

Stored procedures can include detailed exception handling. Functions cannot contain exception handling.

Stored procedures can return multiple resultsets from a single stored procedure call. TVFs can return a single rowset from a function call. There is no mechanism to permit the return of multiple rowsets from a single function call.

- Both can often achieve similar outcomes
- Some source applications can only call one or the other
- Functions
 - Can have their output consumed more easily in code
 - Can return table output in a variable
 - Cannot have data-related side effects
 - Often cause significant performance issues when they are multistatement functions
- Stored procedures
 - Can alter the data
 - Can execute dynamic SQL statements
 - Can include detailed exception handling
 - Can return multiple result sets

Comparing Table-Valued Functions and Views

TVFs can provide similar outcomes to views.

Views and TVFs that do not contain parameters can usually be consumed by most client applications that access tables. Not all such applications can pass parameters to a TVF.

You can update views and inline TVFs. This is not the case for multistatement TVFs.

Views can have INSTEAD OF triggers associated with them. This is mostly used to provide for updatable views based on multiple base tables.

Views and inline TVFs are incorporated into surrounding queries. Multistatement TVFs are not incorporated into surrounding queries, and often lead to performance issues when they are used inappropriately.

- Both can often achieve similar outcomes
- Views
 - Can be consumed by almost all applications
 - Are very similar to tables
 - Can be updatable
 - Can have INSTEAD OF triggers associated with them
- TVFs
 - Are like parameterized views
 - Can often lead to significant performance problems
 - Can be updatable when inline
- Avoid multistatement TVFs if there is any option to apply the same logic inline

Check Your Knowledge

Question	
<p>What is wrong with this TVF code fragment?</p> <pre>SELECT * FROM (EXEC dbo.GetCriticalPathNodes);</pre>	
Select the correct answer.	
	Incorrect syntax for a TVF.
	You cannot select from a stored procedure in a TVF.
	dbo.GetCriticalPathNodes does not exist.
	The statement needs more parentheses.
	None of these—this code is good.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
True or false? You can update views, inline TVFs, and multistatement TVFs.	

Lab: Designing and Implementing User-Defined Functions

Scenario

The existing marketing application includes some functions. Your manager has requested your assistance in creating a new function for formatting phone numbers—you also need to modify an existing function to improve its usability.

Objectives

After completing this lab, you will be able to:

- Create a function.
- Modify an existing function.

Estimated Time: 30 minutes

Virtual machines: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Format Phone Numbers

Scenario

Your manager has noticed that different users tend to format phone numbers that are entered into the database in different ways. She has asked you to create a function that will be used to format the phone numbers. You need to design, implement, and test the function.

The main tasks for this exercise are as follows:

- Review the design requirements.
- Design and create the function.
- Test the function.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Review the Design Requirements
3. Design and Create the Function
4. Test the Function

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **D:\Labfiles\Lab10\Starter\Setup.cmd** as an administrator.

► Task 2: Review the Design Requirements

1. Open the **Supporting Documentation.docx** in the **D:\Labfiles\Lab10\Starter** folder.
2. Review the **Function Specifications: Phone Number** section in the supporting documentation.

► Task 3: Design and Create the Function

- Design and create the function for reformatting phone numbers.

► **Task 4: Test the Function**

- Execute the **FormatPhoneNumber** function to ensure that the function correctly formats the phone number.

Results: After this exercise, you should have created a new **FormatPhoneNumber** function within the **dbo** schema.

Exercise 2: Modify an Existing Function

Scenario

An existing function, **dbo.StringListToTable**, takes a comma-delimited list of strings and returns a table. In some application code, this causes issues with data types because the list often contains integers rather than just strings.

The main tasks for this exercise are as follows:

1. Review the requirements.
2. Design and create the function.
3. Test the function.
4. Test the function by using an alternate delimiter, such as the pipe character (|).

The main tasks for this exercise are as follows:

1. Review the Requirements
2. Design and Create the Function
3. Test the Function
4. Test the Function by Using an Alternate Delimiter

► **Task 1: Review the Requirements**

- In the **Supporting Documentation.docx**, review the **Requirements: Comma Delimited List Function** section in the supporting documentation.

► **Task 2: Design and Create the Function**

- Design and create the **dbo.IntegerListToTable** function.

► **Task 3: Test the Function**

- Execute the **dbo.IntegerListToTable** function to ensure that it returns the correct results.

► **Task 4: Test the Function by Using an Alternate Delimiter**

- Test the **dbo.IntegerListToTable** function, and then pass in an alternate delimiter, such as the pipe character (|).

Results: After this exercise, you should have created a new **IntegerListToTable** function within a **dbo** schema.

Module Review and Takeaways

In this module, you have learned about designing, creating, deploying, and testing functions.



Best Practice: When working with functions, consider the following best practices:

- Avoid calling multistatement TVFs for each row of a query. In many cases, you can dramatically improve performance by extracting the code from the query into the surrounding query.
- Use the WITH EXECUTE AS clause to override the security context of code that needs to perform actions that the user who is executing the code does not have.

Review Question(s)

Question: When you are using the EXECUTE AS clause, what privileges should you grant to the login or user that is being impersonated?

Question: When you are using the EXECUTE AS clause, what privileges should you grant to the login or user who is creating the code?

Module 11

Responding to Data Manipulation Via Triggers

Contents:

Module Overview	11-1
Lesson 1: Designing DML Triggers	11-2
Lesson 2: Implementing DML Triggers	11-9
Lesson 3: Advanced Trigger Concepts	11-15
Lab: Responding to Data Manipulation by Using Triggers	11-23
Module Review and Takeaways	11-26

Module Overview

Data Manipulation Language (DML) triggers are powerful tools that you can use to enforce domain, entity, referential data integrity and business logic. The enforcement of integrity helps you to build reliable applications. In this module, you will learn what DML triggers are, how they enforce data integrity, the different types of trigger that are available to you, and how to define them in your database.

Objectives

After completing this module, you will be able to:

- Design DML triggers.
- Implement DML triggers.
- Explain advanced DML trigger concepts, such as nesting and recursion.

Lesson 1

Designing DML Triggers

Before you begin to create DML triggers, you should become familiar with best practice design guidelines, which help you to avoid making common errors.

Several types of DML trigger are available—this lesson goes through what they do, how they work, and how they differ from Data Definition Language (DDL) triggers. DML triggers have to be able to work with both the previous state of the database and its changed state. You will see how the inserted and deleted virtual tables provide that capability.

DML triggers are often added after applications are built—so it's important to check that a trigger will not cause errors in the existing applications. The SET NOCOUNT ON command helps to avoid the side effects of triggers.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe DML triggers.
- Explain how AFTER triggers differ from INSTEAD OF triggers, and where each should be used.
- Access both the “before” and “after” states of the data by using the inserted and deleted virtual tables.
- Avoid affecting existing applications by using SET NOCOUNT ON.
- Describe performance-related considerations for triggers.

What Are DML Triggers?

A DML trigger is a special kind of stored procedure that executes when an INSERT, UPDATE, or DELETE statement modifies the data in a specified table or view. This includes any INSERT, UPDATE, or DELETE statement that forms part of a MERGE statement. A trigger can query other tables and include complex Transact-SQL statements.

DDL triggers are similar to DML triggers, but DDL triggers fire when DDL events occur. DDL events occur for most CREATE, ALTER, or DROP statements in the Transact-SQL language.

Logon triggers are a special form of trigger that fire when a new session is established. (There is no logoff trigger.)

- Triggers are special stored procedures which:
 - Fire for INSERT, UPDATE, or DELETE DML operations
 - Fire on DDL statements such as CREATE, ALTER, or DROP
 - Provide complex logic and meaningful error messages
 - Multiple triggers can be fired



Note: Terminology: The word “fire” is used to describe the point at which a trigger is executed as the result of an event.

Trigger Operation

The trigger and the statement that fires it are treated as a single operation, which you can roll back from within the trigger. By rolling back an operation, you can undo the effect of a Transact-SQL statement if the logic in your triggers determines that the statement should not have been executed. If the statement is part of another transaction, the outer transaction is also rolled back.

Triggers can cascade changes through related tables in the database; however, in many cases, you can execute these changes more efficiently by using cascading referential integrity constraints.

Complex Logic and Meaningful Error Messages

Triggers can guard against malicious or incorrect INSERT, UPDATE, and DELETE operations and enforce other restrictions that are more complex than those that are defined by using CHECK constraints. For example, a trigger could check referential integrity for one column, only when another column holds a specific value.

Unlike CHECK constraints, triggers can reference columns in other tables. For example, a trigger can use a SELECT statement from another table to compare to the inserted or updated data, and to perform additional actions, such as modifying the data or displaying a user-defined error message.

Triggers can evaluate the state of a table before and after a data modification, and take actions based on that difference. For example, you may want to check that the balance of a customer's account does not change by more than a certain amount if the person processing the change is not a manager.

With triggers, you can also create custom error messages for when constraint violations occur. This could make the messages that are passed to users more meaningful.

Multiple Triggers

With multiple triggers of the same type (INSERT, UPDATE, or DELETE) on a table, you can make multiple different actions occur in response to the same modification statement. You might create multiple triggers to separate the logic that each performs, but note that you do not have complete control over the order in which they fire. You can only specify which triggers should fire first and last.

For more information about triggers, see Microsoft Docs:



CREATE TRIGGER (Transact-SQL)

<https://aka.ms/Bghmb0>

AFTER Triggers vs. INSTEAD OF Triggers

There are two types of DML trigger: AFTER triggers and INSTEAD OF triggers. The main difference between them relates to when they fire. You can implement both types of DML trigger in either Transact-SQL or managed code. In this module, you will explore how they are designed and implemented by using Transact-SQL.

Even if an UPDATE statement (or other data modification statement) modifies many rows, the trigger only fires a single time. For that reason, you must design triggers to handle multiple rows. This design differs from other database engines

Two types of triggers can be implemented in managed code or Transact-SQL:

- AFTER triggers
 - Fire after the event to which they relate
 - Are treated as part of the same transaction as the statement that triggered them
 - Can roll back the statement that triggered them (and any transaction of which that statement was part)
- INSTEAD OF triggers
 - Make it possible to execute alternate code, unlike a BEFORE trigger in other database engines
 - Are often used to create updatable views with more than one base table

where triggers are written to target single rows and are called multiple times when a statement affects multiple rows.

AFTER Triggers

AFTER triggers fire after the data modifications, which are part of the event to which they relate, complete. This means that an INSERT, UPDATE, or DELETE statement executes and modifies the data in the database. After that modification has completed, AFTER triggers associated with that event fire—but still within the same operation that triggered them.

Common reasons for implementing AFTER triggers are:

- To provide auditing of the changes that were made.
- To implement complex rules involving the relationship between tables.
- To implement default values or calculated values within rows.

In many cases, you can replace trigger-based code with other forms of code. For example, Microsoft® SQL Server® data management software might provide auditing. Relationships between tables are more typically implemented by using foreign key constraints. Default values and calculated values are typically implemented by using DEFAULT constraints and persisted calculated columns. However, in some situations, the complexity of the logic that is required will make triggers a good solution.

If the trigger executes a ROLLBACK statement, the data modification statement with which it is associated will be rolled back. If that statement was part of a larger transaction, that outer transaction would be rolled back, too.

INSTEAD OF Triggers

An INSTEAD OF trigger is a special type of trigger that executes alternate code instead of executing the statement from which it was fired.

When you use an INSTEAD OF trigger, only the code in the trigger is executed. The original INSERT, UPDATE, or DELETE operation that caused the trigger to fire does not occur.

INSTEAD OF triggers are most commonly used to make views that are based on multiple base tables updatable.

Inserted and Deleted Virtual Tables

When you design a trigger, you must be able to make decisions based on what changes have been made to the data. To make effective decisions, you need to access details of both the unmodified and modified versions of the data. DML triggers provide this through a pair of virtual tables called "inserted" and "deleted". These virtual tables are often then joined to the modified table data as part of the logic within the trigger.

- Inserted and deleted virtual tables
 - Provide access to state of the data before and after the modification began
 - Are often joined to the modified table data
 - Are available in both AFTER and INSTEAD OF triggers
 - Deleted table
 - DELETE statements – rows just deleted
 - UPDATE statements – original row contents
 - Inserted table
 - INSERT statements – rows just inserted
 - UPDATE statements – modified row contents

INSERT, UPDATE, and DELETE Operations

The following list describes what happens after each of these operations:

- **INSERT:** the inserted virtual table holds details of the rows that have just been inserted. The underlying table also contains those rows.
- **UPDATE:**
 - The inserted virtual table holds details of the modified versions of the rows. The underlying table also contains those rows in the modified form.
 - The deleted virtual table holds details of the rows from before the modification was made. The underlying table holds the modified versions.
- **DELETE:** the deleted virtual table holds details of the rows that have just been deleted. The underlying table no longer contains those rows.

INSTEAD OF Triggers

An INSTEAD OF trigger can be associated with an event on a table. When you attempt an INSERT, UPDATE, or DELETE statement that triggers the event, the inserted and deleted virtual tables hold details of the modifications that must be made, but have not yet happened.

Scope of Inserted and Deleted

The inserted and deleted virtual tables are only available during the execution of the trigger code and are scoped directly to the trigger code. This means that, if the trigger code were to execute a stored procedure, that stored procedure would not have access to the inserted and deleted virtual tables.

SET NOCOUNT ON

When you are adding a trigger to a table, you must avoid affecting the behavior of applications that are accessing the table, unless the intended purpose of the trigger is to prevent misbehaving applications from making inappropriate data changes.

It is common for application programs to issue data modification statements and to check the returned count of the number of rows that are affected.

This process is often performed as part of an optimistic concurrency check.

Consider the following code example:

UPDATE Statement

```
UPDATE Customer
SET Customer.FullName = @NewName,
Customer.Address = @NewAddress
WHERE Customer.CustomerID = @CustomerID
AND Customer.Concurrency = @Concurrency;
```

- Triggers should not return rows of data
- Client applications often check the number of rows that are affected by data modification statements
- Triggers that are affected by data modification statements
- SET NOCOUNT ON avoids affecting the outer statements
- Returning rowsets has been deprecated
- Use the configuration setting "Disallow results from triggers" to prevent triggers from returning resultsets

In this case, the **Concurrency** column is a rowversion data type column. The application was designed so that the update only occurs if the **Concurrency** column has not been altered. Using rowversion columns, every modification to the row causes a change in the rowversion column.

When the application intends to modify a single row, it issues an UPDATE statement for that row. The application then checks the count of updated rows that are returned by SQL Server. When the application sees that only a single row has been modified, the application knows that only the row that it intended to change was affected. It also knows that no other user had modified the row before the application read the data.

A common mistake when you are adding triggers is that if the trigger also causes row modifications (for example, writes an audit row into an audit table), that count is returned in addition to the expected count. You can avoid this situation by using the SET NOCOUNT ON statement. Most triggers should include this statement.



Note: Returning Rowsets

Although you can include a SELECT statement within a trigger and for it to return rows, the creation of this type of side effect is discouraged. The ability to do this is now deprecated and should not be used in new development work. There is a configuration setting, "Disallow results from triggers", which, when it is set to 1, disallows this capability.

Considerations for Triggers

For performance reasons, it is generally preferable to use constraints rather than triggers. Triggers are complex to debug because the actions that they perform are not visible directly in the code that causes them to fire. Triggers also increase how long data modification transactions take, because they add extra steps that SQL Server needs to process during these operations. You should design triggers to be as short as possible and to be specific to a given task, rather than being designed to perform a large number of tasks within a single trigger.

- Constraints:
 - Are preferred to triggers
 - Avoid data modification overhead on violation
- Triggers:
 - Are complex to debug
 - Use a rowversion store in **tempdb** database
 - Excessive usage can impact **tempdb** performance
 - Can increase the duration of transactions
 - Managing Trigger Security

Note that you can disable and re-enable triggers by using the ALTER TRIGGER statement.

Constraints vs. Triggers

When an AFTER trigger decides to disallow a data modification, it does so by executing a ROLLBACK statement. The ROLLBACK statement undoes all of the work that the original statement performed. However, you can achieve higher performance by avoiding the data modification ever occurring.



Note: Reminder: Constraints are rules that define allowed column or table values.

Constraints are checked before any data modification is attempted, so they often provide much higher performance than is possible with triggers, particularly in ROLLBACK situations. You can use constraints for relatively simple checks; triggers make it possible to check more complex logic.

Rowversions and tempdb

Since SQL Server 2005, trigger performance has been improved. In earlier versions of SQL Server, the inserted and deleted virtual tables were constructed from entries in the transaction log. From SQL Server 2005 onward, a special rowversion table has been provided in the **tempdb** database. The rowversion table holds copies of the data in the inserted and deleted virtual tables for the duration of the trigger. This design has improved overall performance, but means that excessive usage of triggers could cause performance issues within the **tempdb** database.

Managing Trigger Security

When a DML or DDL trigger is executed, it is executed according to the calling user context. Context refers to the user privilege—for example, **sysadmin** (fixed role) or **ADVENTUREWORKS\Student** (user-defined role).

The default context can be a security issue because it could be used by people who wish to execute malicious code.

The following example shows how to change permissions for Student to sysadmin:

Change Permissions DDL Example

```
CREATE TRIGGER DDL_trigStudent
ON DATABASE
FOR ALTER_TABLE
AS
GRANT CONTROL SERVER TO Student;
GO
```

The **Student** user now has CONTROL SERVER permissions. **Student** has been able to grant permission that she/he could not have normally done. The code has granted escalated permissions.

 **Best Practice:** To prevent triggers from firing under escalated privileges, you should first understand what triggers you have in the database and server instance. You can use the **sys.triggers** and **sys.server_triggers** views to find out. Secondly, use the **DISABLE_TRIGGER** statement to disable triggers that use escalated privileges.

For further information about managing triggers, see Microsoft Docs:

Manage Trigger Security

<http://aka.ms/nzsq10>

Check Your Knowledge

Question
Which types of statements fire DML triggers?
Select the correct answer.
<input type="checkbox"/> CREATE, ALTER, DROP
<input type="checkbox"/> MERGE, ALTER, DELETE
<input type="checkbox"/> MERGE, CREATE, INSERT
<input type="checkbox"/> CREATE, UPDATE, DELETE
<input type="checkbox"/> DELET, INSERT, UPDATE

Question: What reasons can you think of for deploying AFTER triggers?

Lesson 2

Implementing DML Triggers

The first lesson provided information about designing DML triggers. We now consider how to implement the designs that have been created.

Lesson Objectives

After completing this lesson, you will be able to:

- Implement AFTER INSERT triggers.
- Implement AFTER DELETE triggers.
- Implement AFTER UPDATE triggers.

AFTER INSERT Triggers

An AFTER INSERT trigger executes whenever an INSERT statement enters data into a table or view on which the trigger is configured. The action of the INSERT statement is completed before the trigger fires, but the trigger action is logically part of the INSERT operation.

- INSERT statement is executed
- AFTER INSERT trigger then fires
- Ensures that multirow inserts are supported

AFTER INSERT Trigger Actions

When an AFTER INSERT trigger fires, new rows are added to both the base table and the inserted virtual table. The inserted virtual table holds a copy of the rows that have been inserted into the base table.

The trigger can examine the inserted virtual table to determine what to do in response to the modification.

Multirow Inserts

In the code example on the slide, insertions for the **Sales.Opportunity** table are being audited to a table called **Sales.OpportunityAudit**. Note that the trigger processes all inserted rows at the same time. A common error when designing AFTER INSERT triggers is to write them with the assumption that only a single row is being inserted.

Example code that works with multiple rows:

CREATE TRIGGER

```
CREATE TRIGGER Sales.InsertCustomer
    ON Sales.Customer
    AFTER INSERT
    AS
    BEGIN
        SET NOCOUNT ON;
        INSERT INTO Sales.Customer (CustomerID, PersonID, StoreID, TerritoryID)
            SELECT CustomerID, PersonID, StoreID, TerritoryID
            FROM inserted;
    END;
GO
```

For more information about DML Trigger, see Microsoft Docs:



DML Triggers

<https://aka.ms/xqgne8>

Demonstration: Working with AFTER INSERT Triggers

In this demonstration, you will see how to:

- Create an AFTER INSERT trigger.

Demonstration Steps

Create an AFTER INSERT Trigger

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **D:\Demofiles\Mod11\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. On the taskbar, click **Microsoft SQL Server Management Studio**.
5. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, and then click **Connect**.
6. On the **File** menu, point to **Open**, click **Project/Solution**.
7. In the **Open Project** dialog box, navigate to **D:\Demofiles\Mod11\Demo11**, click **Demo11.ssmssqlproj**, and then click **Open**.
8. In Solution Explorer, expand the **Queries** folder, and then double-click the **21 - Demonstration 2A.sql** script file to open it.
9. Select the code under the **Step A** comment, and then click **Execute**.
10. Select the code under the **Step B** comment, and then click **Execute**.
11. Select the code under the **Step C** comment, and then click **Execute**.
12. Select the code under the **Step D** comment, and then click **Execute**.
13. Select the code under the **Step E** comment, and then click **Execute**. Note the error message.
14. Select the code under the **Step F** comment, and then click **Execute**.
15. Do not close SQL Server Management Studio.

AFTER DELETE Triggers

An AFTER DELETE trigger executes whenever a DELETE statement removes data from a table or view on which the trigger is configured. The action of the DELETE statement is completed before the trigger fires, but logically within the operation of the statement that fired the trigger.

- DELETE statement is executed
- AFTER DELETE trigger then fires
- Multirow deletes
- Truncate table

AFTER DELETE Trigger Actions

When an AFTER DELETE trigger fires, rows are removed from the base table. The deleted virtual table holds a copy of the rows that have been deleted from the base table. The trigger can examine the deleted virtual table to determine what to do in response to the modification.

Multirow Deletes

In the code example, rows in the **Production.Product** table are being flagged as discontinued if the product subcategory row with which they are associated in the **Production.SubCategory** table is deleted. Note that the trigger processes all deleted rows at the same time. A common error when designing AFTER DELETE triggers is to write them with the assumption that only a single row is being deleted.

Setting the Discontinued Date with a Trigger if a Subcategory is Deleted

```
CREATE TRIGGER TR_Category_Delete
ON Production.ProductSubCategory
AFTER DELETE AS
BEGIN
    SET NOCOUNT ON;
    UPDATE p set p.DiscontinuedDate=SYSDATETIME()
    FROM Production.Product AS p
    WHERE EXISTS (SELECT 1 FROM deleted AS d
    WHERE p.ProductSubcategoryID =d.ProductSubcategoryID);
END;
GO
```

TRUNCATE TABLE

When rows are deleted from a table by using a DELETE statement, any AFTER DELETE triggers are fired when the deletion is completed. TRUNCATE TABLE is an administrative option that removes all rows from a table. It needs additional permissions above those required for deleting rows. It does not fire any AFTER DELETE triggers that are associated with the table.

Demonstration: Working with AFTER DELETE Triggers

In this demonstration, you will see how to:

- Create and test AFTER DELETE triggers.

Demonstration Steps

Create and Test AFTER DELETE Triggers

1. In Solution Explorer, in the **Queries** folder, and then double-click the **22 - Demonstration 2B.sql** script file to open it.
2. Select the code under the **Step A** comment, and then click **Execute**.
3. Select the code under the **Step B** comment, and then click **Execute**.
4. Select the code under the **Step C** comment, and then click **Execute**.
5. Select the code under the **Step D** comment, and then click **Execute**. Note the error message.
6. Select the code under the **Step E** comment, and then click **Execute**.
7. Do not close SQL Server Management Studio.

AFTER UPDATE Triggers

An AFTER UPDATE trigger executes whenever an UPDATE statement modifies data in a table or view on which the trigger is configured. The action of the UPDATE statement is completed before the trigger fires.

- UPDATE statement is executed
- AFTER UPDATE trigger then fires
- Trigger processes updated rows at the same time

AFTER UPDATE Trigger Actions

When an AFTER UPDATE trigger fires, update actions are treated as a set of deletions of how the rows were and insertions of how the rows are now. Rows that are to be modified in the base table are copied to the deleted virtual table, and the updated versions of the rows are copied to the inserted virtual table. The inserted virtual table holds a copy of the rows in their modified state, the same as how the rows appear now in the base table.

The trigger can examine both the inserted and deleted virtual tables to determine what to do in response to the modification.

Multirow Updates

In the code example, the **Product.ProductReview** table contains a column called **ModifiedDate**. The trigger is being used to ensure that when changes are made to the **Product.ProductReview** table, the value in the **ModifiedDate** column reflects when any changes last happened.

Note that the trigger processes all updated rows at the same time. A common error when designing AFTER UPDATE triggers is to write them with the assumption that only a single row is being updated.

After Update Trigger

```
CREATE TRIGGER TR_ProductReview_Update
ON Production.ProductReview
AFTER UPDATE AS
BEGIN
    SET NOCOUNT ON;
    UPDATE pr
    SET pr.ModifiedDate = SYSDATETIME()
    FROM Production.ProductReview AS pr
    INNER JOIN inserted as i
    ON i.ProductReviewID=pr.ProductReviewID;
END;
```

Demonstration: Working with AFTER UPDATE Triggers

In this demonstration, you will see how to:

- Create and test AFTER UPDATE triggers.

Demonstration Steps

Create and Test AFTER UPDATE Triggers

1. In Solution Explorer, in the **Queries** folder, double-click **23 - Demonstration 2C.sql** to open it.
2. Select the code under the **Step A** comment, and then click **Execute**.
3. Select the code under the **Step B** comment, and then click **Execute**.
4. Select the code under the **Step C** comment, and then click **Execute**.
5. Select the code under the **Step D** comment, and then click **Execute**.
6. Select the code under the **Step E** comment, and then click **Execute**.
7. Select the code under the **Step F** comment, and then click **Execute**.
8. Select the code under the **Step G** comment, and then click **Execute**.
9. Select the code under the **Step H** comment, and then click **Execute**.
10. Select the code under the **Step I** comment, and then click **Execute**.
11. Select the code under the **Step J** comment, and then click **Execute**.
12. Select the code under the **Step K** comment, and then click **Execute**. Note that no triggers are returned.
13. Do not close SQL Server Management Studio.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
Is the following statement true or false? "When rows are deleted from a table by using a DELETE statement, any AFTER DELETE triggers are fired when the deletion is completed. DELETE ROWS is an administrative option that removes all rows from a table."	

Question:

Analyze this create trigger code and indicate the four errors. You can assume the table and columns have been created.

Outline some code you could use to test the trigger.

```
CREATE TRIGGER TR_SellingPrice_InsertUpdate
IN dbo.SellingPrice
AFTER INPUT, UPDATE AS BEGIN
    SET NOCOUNT OPEN;
    INSERT sp
    SET sp.ExtendedAmount = sp.SubTotal
        + sp.TaxAmount
        + sp.FreightAmount
    FROM dbo.SellingPrice AS sp
    INNER JOIN inserted AS i
    ON sp.SellingPriceID = i.SellingPriceID;
END;
GO
```

Lesson 3

Advanced Trigger Concepts

In the previous two lessons, you have learned to design and implement DML AFTER triggers. However, to make effective use of these triggers, you have to know and understand some additional areas of complexity that are related to them. This lesson considers when to use triggers, and when to consider alternatives.

Lesson Objectives

After completing this lesson, you will be able to:

- Implement DML INSTEAD OF triggers.
- Explain how nested triggers work and how configurations might affect their operation.
- Explain considerations for recursive triggers.
- Use the UPDATE function to build logic based on the columns being updated.
- Describe the order in which multiple triggers fire when defined on the same object.
- Explain the alternatives to using triggers.

INSTEAD OF Triggers

INSTEAD OF triggers cause the execution of alternate code instead of executing the statement that caused them to fire.

INSTEAD OF Triggers vs. BEFORE Triggers

Some other database engines provide BEFORE triggers. In those databases, the action in the BEFORE trigger happens before the data modification statement that also occurs. INSTEAD OF triggers in SQL Server are different from the BEFORE triggers that you may have encountered in other database engines. Using an INSTEAD OF trigger as it is implemented in SQL Server, only the code in the trigger is executed. The original operation that caused the trigger to fire is not executed.

- INSERT, UPDATE or DELETE statement requested to be executed
- Statement does not execute
- INSTEAD OF trigger code executes instead
- Updatable views are a common use

Updatable Views

You can define INSTEAD OF triggers on views that have one or more base tables, where they can extend the types of updates that a view can support.

This trigger executes instead of the original triggering action. INSTEAD OF triggers increase the variety of types of updates that you can perform against a view. Each table or view is limited to one INSTEAD OF trigger for each triggering action (INSERT, UPDATE, or DELETE).

You can specify an INSTEAD OF trigger on both tables and views. You cannot create an INSTEAD OF trigger on views that have the WITH CHECK OPTION clause defined. You can perform operations on the base tables within the trigger. This avoids the trigger being called again. For example, you could perform a set of checks before inserting data, and then perform the insert on the base table.

INSTEAD OF Trigger

```
CREATE TRIGGER TR_ProductReview_Delete
ON Production.ProductReview
INSTEAD OF DELETE AS
BEGIN
    SET NOCOUNT ON;
    UPDATE pr set pr.ModifiedDate = SYSDATETIME()
    FROM Production.ProductReview AS pr
    INNER JOIN deleted as d
    ON pr.ProductReviewID = d.ProductReviewID;
END;
```

Demonstration: Working with INSTEAD OF Triggers

In this demonstration, you will see how to:

- Create and test an INSTEAD OF DELETE trigger.

Demonstration Steps

Create and Test an INSTEAD OF DELETE Trigger

1. In Solution Explorer, in the **Queries** folder, double-click **31 - Demonstration 3A.sql** in the **Solution Explorer** script file to open it.
2. Select the code under the **Step A** comment, and then click **Execute**.
3. Select the code under the **Step B** comment, and then click **Execute**.
4. Select the code under the **Step C** comment, and then click **Execute**.
5. Select the code under the **Step D** comment, and then click **Execute**.
6. Select the code under the **Step E** comment, and then click **Execute**.
7. Select the code under the **Step F** comment, and then click **Execute**.
8. Select the code under the **Step G** comment, and then click **Execute**.
9. Select the code under the **Step H** comment, and then click **Execute**.
10. Select the code under the **Step I** comment, and then click **Execute**.
11. Select the code under the **Step J** comment, and then click **Execute**.
12. Select the code under the **Step K** comment, and then click **Execute**.
13. Select the code under the **Step L** comment, and then click **Execute**. Note the error message.
14. Select the code under the **Step M** comment, and then click **Execute**. Note the error message.
15. Select the code under the **Step N** comment, and then click **Execute**.
16. Select the code under the **Step O** comment, and then click **Execute**.
17. Select the code under the **Step P** comment, and then click **Execute**.
18. Select the code under the **Step R** comment, and then click **Execute**.
19. Select the code under the **Step S** comment, and then click **Execute**.
20. Select the code under the **Step U** comment, and then click **Execute**.
21. Select the code under the **Step V** comment, and then click **Execute**.

22. Do not close SQL Server Management Studio.

How Nested Triggers Work

Triggers can contain UPDATE, INSERT, or DELETE statements. When these statements on one table cause triggers on another table to fire, the triggers are considered to be nested.

Triggers are often used for auditing purposes. Nested triggers are essential for full auditing to occur. Otherwise, actions would occur on tables without being audited.

You can control whether nested trigger actions are permitted. By default, these actions are permitted by using a configuration option at the server level. You can also detect the current nesting level by querying `@@nestlevel`.

A failure at any level of a set of nested triggers cancels the entire original statement, and all data modifications are rolled back.

A nested trigger will not fire twice in the same trigger transaction; a trigger does not call itself in response to a second update to the same table within the trigger.

- Turned on at the server level
- Complex to debug

Complexity of Debugging

We noted in an earlier lesson that debugging triggers can sometimes be difficult. Nested triggers are particularly difficult to debug. One common method that is used during debugging is to include PRINT statements within the body of the trigger code so that you can determine where a failure occurred. However, you should make sure these statements are only used during debugging phases.

Considerations for Recursive Triggers

A recursive trigger is a trigger that performs an action which causes the same trigger to fire again either directly or indirectly. Any trigger can contain an UPDATE, INSERT, or DELETE statement that affects the same table or another table. By switching on the recursive trigger option on a database, a trigger that changes data in a table can start itself again, in a recursive execution.

Direct Recursion

Direct recursion occurs when a trigger fires and performs an action on the same table that causes the same trigger to fire again.

For example, an application updates table **T1**, which causes trigger **Trig1** to fire. Trigger **Trig1** updates table **T1** again, which causes trigger **Trig1** to fire again.

- Recursive triggers are disabled by default
- To turn them on:
 - ALTER DATABASE *db* SET RECURSIVE_TRIGGERS ON
 - Direct Recursion
 - Indirect Recursion
 - Considerations:
 - Careful design and testing to ensure that the 32-level nesting limit is not exceeded
 - Difficult to control the order of table updates
 - Can usually be replaced by nonrecursive logic
 - The RECURSIVE TRIGGERS option only affects direct recursion

Indirect Recursion

Indirect recursion occurs when a trigger fires and performs an action that causes another trigger to fire on a different table which, in turn, causes an update to occur on the original table which, in turn, causes the original trigger to fire again.

For example, an application updates table **T2**, which causes trigger **Trig2** to fire. **Trig2** updates table **T3**, which causes trigger **Trig3** to fire. In turn, trigger **Trig3** updates table **T2**, which causes trigger **Trig2** to fire again.

To prevent indirect recursion of this sort, turn off the nested triggers option at the server instance level.

Recursive Triggers Considerations

The following list provides considerations for recursive triggers:

- Careful design and thorough testing is required to ensure that the 32-level nesting limit is not exceeded.
- Can be difficult to control the order of table updates.
- Can usually be replaced by nonrecursive logic.
- The RECURSIVE_TRIGGERS option only affects direct recursion.

UPDATE Function

It is a common requirement to build logic that only takes action if particular columns are being updated.

You can use the UPDATE function to detect whether a particular column is being updated in the action of an UPDATE statement. For example, you might want to take a particular action only when the size of a product changes. The column is referenced by the name of the column. The Update function is used in the AFTER INSERT and AFTER UPDATE triggers.

- UPDATE function—is a column being updated?
- Used in AFTER INSERT and AFTER UPDATE
- COLUMNS_UPDATED function returns bitmap of columns being updated



Note: Function or Statement? Be careful not to confuse the UPDATE function with the UPDATE statement.

Change of Value

Note that the UPDATE function does not indicate if the value is actually changing. It only indicates if the column is part of the list of columns in the SET clause of the UPDATE statement. To detect if the value in a column is actually being changed to a different value, you must interrogate the inserted and deleted virtual tables.

COLUMNS_UPDATED Function

SQL Server also provides a function called COLUMNS_UPDATED. This function returns a bitmap that indicates which columns are being updated. The values in the bitmap depend upon the positional information for the columns. Hard-coding that sort of information in the code within a trigger is generally not considered good coding practice because it affects the readability—and therefore the maintainability—of your code. It also reduces the reliability of your code because schema changes to the table could break it.

In this example, the trigger uses the UPDATE function to identify updates to a particular column:

Update Function

```
CREATE TRIGGER pdate_ListPriceAudit
    ON Production.Product AFTER UPDATE AS
BEGIN
    IF UPDATE(ListPrice)
        BEGIN
            INSERT INTO Production.ListPriceAudit (ProductID, ListPrice,
ChangedWhen)
                SELECT i.ProductID, i.ListPrice, SYSDATETIME()
            FROM inserted AS i;
        END;
    END;
```

Firing Order for Triggers

You can assign multiple triggers to a single event on a single object. Only limited control is available over the firing order of these triggers.

sp_settriggerorder

Developers often want to control the firing order of multiple triggers that are defined for a single event on a single object. For example, a developer might create three AFTER INSERT triggers on the same table, each implementing different business rules or administrative tasks.

In general, code within one trigger should not depend upon the order of execution of other triggers. Limited control of firing order is available through the **sp_settriggerorder** system stored procedure. With **sp_settriggerorder**, you can specify the triggers that will fire first and last from a set of triggers that all apply to the same event, on the same object.

The possible values for the **@order parameter** are **First**, **Last**, or **None**. **None** is the default action. An error will occur if the First and Last triggers both refer to the same trigger.

For DML triggers, the possible values for the **@stmttype** parameter are **INSERT**, **UPDATE**, or **DELETE**.

Use **sp_settriggerorder** to set the sequence that a trigger fires.

- Multiple triggers may be created for a single event
- You cannot specify the order in which the triggers will fire
- With **sp_settriggerorder**, you can specify which triggers will fire first and last

Sp_settriggerorder

```
EXEC sp_settriggerorder
    @triggername = 'Production.TR_Product_Update_ListPriceAudit',
    @order = 'First',
    @stmttype = 'UPDATE';
```

Alternatives to Triggers

Triggers are useful in many situations, and are sometimes necessary to handle complex logic. However, triggers are sometimes used in situations where alternatives might be preferable.

Checking Values

You could use triggers to check that values in columns are valid or within given ranges. However, in general, you should use CHECK constraints instead—CHECK constraints perform this validation before the data modification is attempted.

If you are using triggers to check the correlation of values across multiple columns within a table, you should generally create table-level CHECK constraints instead.

Many developers use triggers in situations where alternatives would be preferable:

- Use constraints for checking values
- Use defaults for values not supplied during inserts
- Use foreign key constraints to check for referential integrity
- Use computed and persisted computed columns
- Use indexed views for precalculating aggregates

Defaults

You can use triggers to provide default values for columns when no values have been provided in INSERT statements. However, you should generally use DEFAULT constraints for this instead.

Foreign Keys

You can use triggers to check the relationship between tables. However, you should generally use FOREIGN KEY constraints for this.

Computed Columns

You can use triggers to maintain the value in one column based on the value in other columns. In general, you should use computed columns or persisted computed columns for this.

Precalculating Aggregates

You can use triggers to maintain precalculated aggregates in one table, based on the values in rows in another table. In general, you should use indexed views to provide this functionality.

Suitable Situations for Using Triggers

Although general guidelines are provided here, replacing the triggers with these alternatives is not always possible. For example, the logic that is required when checking values might be too complex for a CHECK constraint.

As another example, a FOREIGN KEY constraint cannot be contained on a column that is also used for other purposes. Consider a column that holds an employee number only if another column holds the value “E”. This typically indicates a poor database design, but you can use triggers to ensure this sort of relationship.

Demonstration: Replacing Triggers with Computed Columns

In this demonstration, you will see:

- How to replace a trigger with a computed column.

Demonstration Steps

Replace a Trigger with a Computed Column

1. In Solution Explorer, in the **Queries** folder, double-click **32 - Demonstration 3B.sql** in the **Solution Explorer** script file to open it.
2. Select the code under the **Step A** comment, and then click **Execute**.
3. Select the code under the **Step B** comment, and then click **Execute**.
4. Select the code under the **Step C** comment, and then click **Execute**.
5. Select the code under the **Step D** comment, and then click **Execute**.
6. Select the code under the **Step E** comment, and then click **Execute**.
7. Select the code under the **Step F** comment, and then click **Execute**.
8. Select the code under the **Step G** comment, and then click **Execute**.
9. Close SQL Server Management Studio, without saving any changes.

Check Your Knowledge

Question
<p>What are the four missing terms from this statement indicated by <XXX>?</p> <p><XXX></p> <p>These triggers are most commonly used to enable views that are based on multiple base tables to be updatable. You can define <XXX> triggers on views that have one or more base tables, where they can extend the types of updates that a view can support.</p> <p>This trigger executes instead of the original triggering action. <XXX> triggers increase the variety of types of updates that you can perform against a view. Each table or view is limited to one <xxx> trigger for each triggering action (<XXX>).</p> <p>You can specify an <XXX> trigger on both tables and views. You cannot create an <XXX> trigger on views that have the <XXX> clause defined. You can perform operations on the base tables within the trigger. This avoids the trigger being called again. For example, you could perform a set of checks before inserting data, and then perform the insert on the base table.</p>
Select the correct answer.
<input type="checkbox"/> How Nested Triggers Work; INSTEAD OF; DELETE; CHECK OPTION
<input type="checkbox"/> Updatable Views; AFTER INSERT; INSERT, UPDATE, or DELETE; NO ROWCOUNT
<input type="checkbox"/> Updatable Views; INSTEAD OF; DML or DDL; CHECK OPTION
<input type="checkbox"/> Updatable Views; RATHER THAN; UPDATE; CHECK OPTION
<input type="checkbox"/> Updatable Views; INSTEAD OF; INSERT, UPDATE, or DELETE; CHECK OPTION

Lab: Responding to Data Manipulation by Using Triggers

Scenario

You are required to audit any changes to data in a table that contains sensitive balance data. You have decided to implement this by using DML triggers because the SQL Server Audit mechanism does not provide directly for the requirements in this case.

Supporting Documentation

The **Production.ProductAudit** table is used to hold changes to high value products. The data to be inserted in each column is shown in the following table:

Column	Data type	Value to insert
AuditID	int	IDENTITY
ProductID	int	ProductID
UpdateTime	datetime2	SYSDATETIME()
ModifyingUser	varchar(30)	ORIGINAL_LOGIN()
OriginalListPrice	decimal(18,2)	ListPrice before update
NewListPrice	decimal(18,2)	ListPrice after update

Objectives

After completing this lab, you will be able to:

- Create triggers
- Modify triggers
- Test triggers

Estimated Time: 30 minutes

Virtual Machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Create and Test the Audit Trigger

Scenario

The **Production.Product** table includes a column called **ListPrice**. Whenever an update is made to the table, if either the existing balance or the new balance is greater than **1,000 US dollars**, an entry must be written to the **Production.ProductAudit** audit table.

 **Note:** Inserts or deletes on the table do not have to be audited. Details of the current user can be taken from the `ORIGINAL_LOGIN()` function.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Review the Design Requirements
3. Design a Trigger
4. Test the Behavior of the Trigger

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab11\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.

► Task 2: Review the Design Requirements

1. In the **D:\Labfiles\Lab11\Starter** folder, open the **Supporting Documentation.docx**.
2. In SQL Server Management Studio, review the existing structure of the **Production.ProductAudit** table and the values required in each column, based on the supporting documentation.
3. Review the existing structure of the **Production.Product** table on SSMS.

► Task 3: Design a Trigger

- Design and create a trigger that meets the needs of the supporting documentation.

► Task 4: Test the Behavior of the Trigger

- Execute data modification statements that are designed to test whether the trigger is working as expected.

Results: After this exercise, you should have created a new trigger. Tests should have shown that it is working as expected.

Exercise 2: Improve the Audit Trigger

Scenario

Now that the trigger created in the first exercise has been deployed to production, the operations team is complaining that too many entries are being audited. Many accounts have more than **10,000 US dollars** as a balance and minor movements of money are causing audit entries. You must modify the trigger so that only changes in the balance of more than **10,000 US dollars** are audited instead.

The main tasks for this exercise are as follows:

1. Modify the trigger based on the updated requirements.
2. Delete all rows from the **Marketing.CampaignAudit** table.
3. Test the modified trigger.

The main tasks for this exercise are as follows:

1. Modify the Trigger
2. Delete all Rows from the Marketing.CampaignAudit Table
3. Test the Modified Trigger

► Task 1: Modify the Trigger

1. Review the design of the existing trigger and decide what modifications are required.
2. Use an ALTER TRIGGER statement to change the existing trigger so that it will meet the updated requirements.

► Task 2: Delete all Rows from the Marketing.CampaignAudit Table

- Execute a DELETE statement to remove all existing rows from the **Marketing.CampaignAudit** table.

► Task 3: Test the Modified Trigger

1. Execute data modification statements that are designed to test whether the trigger is working as expected.
2. Close SQL Server Management Studio without saving anything.

Results: After this exercise, you should have altered the trigger. Tests should show that it is now working as expected.

Module Review and Takeaways

 **Best Practice:** In many business scenarios, it makes sense to mark records as deleted with a status column and use a trigger or stored procedure to update an audit trail table. The changes can then be audited, the data is not lost, and the IT staff can perform purges or archival of the deleted records.

Avoid using triggers in situations where constraints could be used instead.

Review Question(s)

Question: How do constraints and triggers differ regarding timing of execution?

Module 12

Using In-Memory Tables

Contents:

Module Overview	12-1
Lesson 1: Memory-Optimized Tables	12-2
Lesson 2: Natively Compiled Stored Procedures	12-11
Lab: Using In-Memory Database Capabilities	12-16
Module Review and Takeaways	12-19

Module Overview

Microsoft® SQL Server® 2014 data management software introduced in-memory online transaction processing (OLTP) functionality features to improve the performance of OLTP workloads. Subsequent versions of SQL Server add several enhancements, such as the ability to alter a memory-optimized table without recreating it. Memory-optimized tables are primarily stored in memory, which provides the improved performance by reducing hard disk access.

Natively compiled stored procedures further improve performance over traditional interpreted Transact-SQL.

Objectives

After completing this module, you will be able to:

- Use memory-optimized tables to improve performance for latch-bound workloads.
- Use natively compiled stored procedures.

Lesson 1

Memory-Optimized Tables

You can use memory-optimized tables as a way to improve the performance of latch-bound OLTP workloads. Memory-optimized tables are stored in memory, and do not use locks to enforce concurrency isolation. This dramatically improves performance for many OLTP workloads.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the key features of memory-optimized tables.
- Describe scenarios for memory-optimized tables.
- Add a filegroup for memory-optimized tables.
- Create memory-optimized tables.
- Use indexes in memory-optimized tables.
- Use the Memory-Optimization Advisor.
- Query memory-optimized tables.

What Are Memory-Optimized Tables?

Memory-optimized tables in SQL Server are defined as C structs and compiled as dynamic-link libraries (DLLs) that can be loaded into memory. The query processor in SQL Server transparently converts Transact-SQL queries against memory-optimized tables into the appropriate C calls, so you use them just like any other table in a SQL Server database.

Memory-optimized tables:

- Are defined as C structs, compiled into DLLs, and loaded into memory.
- Can persist their data to disk as FILESTREAM data, or they can be nondurable.
- Do not apply any locking semantics during transactional data modifications.
- Contain at least one index.
- Can coexist with disk-based tables in the same database.
- Can be queried by using Transact-SQL through interop services that the SQL Server query processor provides.

- Defined as C structs, compiled into DLLs, and loaded into memory
- Can be persisted as FILESTREAM data, or nondurable
- Do not apply any locking semantics
- Contain at least one index
- Can coexist with disk-based tables
- Can be queried by using Transact-SQL
- Support most data types and features

Supported Data Types and Features

Most data types in memory-optimized tables are supported. However, some are not supported, including *text* and *image*.

For more information about the data types that memory-optimized tables support, see the topic *Supported Data Types for In-Memory OLTP* in Microsoft Docs:

 **Supported Data Types for In-Memory OLTP**

<http://aka.ms/jf3ob7>

Memory-optimized tables support most features that disk-based tables support.

For information about features that are not supported, see the "Memory-Optimized Tables" section of the topic *Transact-SQL Constructs Not Supported by In-Memory OLTP* in Microsoft Docs:

 **Transact-SQL Constructs Not Supported by In-Memory OLTP**

<https://aka.ms/Uckvs9>

Scenarios for Memory-Optimized Tables

Memory-optimized tables provide some performance benefits by storing data in the memory and reducing disk I/O. However, SQL Server uses caching to optimize queries that access commonly used data anyway, so the gains from in-memory storage may not be significant for some tables. The primary feature of memory-optimized tables that improves database performance is the lack of any locking to manage transaction isolation. Memory-optimized tables are therefore likely to be of most benefit when you need to optimize performance for latch-bound workloads that support concurrent access to the same tables.

- Optimistic concurrency optimizes latch-bound workloads
- Common scenarios:
 - Multiple concurrent transactions modifying large numbers of rows
 - A table containing "hot" pages
- Applications should handle conflict errors:
 - Write conflicts
 - Repeatable read validation failures
 - Serializable validation failures
 - Commit dependency failures

Common Scenarios for Memory-Optimized Tables

Common latch-bound scenarios include OLTP workloads in which:

- Multiple concurrent queries modify large numbers of rows in a transaction.
- A table contains "hot" pages. For example, a table that contains a clustered index on an incrementing key value will inherently suffer from concurrency issues because all insert transactions occur in the last page of the index.

Considerations for Memory-Optimized Table Concurrency

When you update data in memory-optimized tables, SQL Server uses an optimistic concurrency row-versioning mechanism to track changes to rows, so that the values in a row at a specific time are known. The in-memory nature of memory-optimized tables means that data modifications occur extremely quickly and conflicts are relatively rare. However, if a conflict error is detected, the transaction in which the error occurred is terminated. You should therefore design applications to handle concurrency conflict errors in a similar way to handling deadlock conditions.

Concurrency errors that can occur in memory-optimized tables include:

- **Write conflicts.** These occur when an attempt is made to update or delete a record that has been updated since the transaction began.

- **Repeatable read validation failures.** These occur when a row that the transaction has read has changed since the transaction began.
- **Serializable validation failures.** These occur when a new (or phantom) row is inserted into the range of rows that the transaction accesses while it is still in progress.
- **Commit dependency failures.** These occur when a transaction has a dependency on another transaction that has failed to commit.

Creating a Filegroup for Memory-Optimized Data

Databases in which you want to create memory-optimized tables must contain a filegroup for memory-optimized data.

You can add a filegroup for memory-optimized data to a database by using the ALTER DATABASE statement, as the following example shows:

Adding a Filegroup for Memory-Optimized Data

```
ALTER DATABASE MyDB
ADD FILEGROUP mem_data CONTAINS
MEMORY_OPTIMIZED_DATA;
GO
ALTER DATABASE MyDB
ADD FILE (NAME = 'MemData', FILENAME = 'D:\Data\MyDB_MemData.ndf')
TO FILEGROUP mem_data;
```

- Databases with memory-optimized tables must contain a filegroup for memory-optimized data
- Either:
 - Use Transact-SQL, or
 - Filegroups page of database properties in SSMS

You can also add a filegroup for memory-optimized data to a database on the **Filegroups** page of the **Database Properties** dialog box in SQL Server Management Studio (SSMS).

Creating Memory-Optimized Tables

Use the CREATE TABLE statement with the MEMORY_OPTIMIZED option to create a memory-optimized table.

Durability

When you create a memory-optimized table, you can specify the durability of the table data.

By default, the durability option is set to SCHEMA_AND_DATA. For SCHEMA_AND_DATA durability, the data in the table is persisted to FILESTREAM data in the memory-optimized filegroup on which the table is created. The data is written to disk as a stream, not in 8-KB pages as used by disk-based tables.

- Durability
 - SCHEMA_AND_DATA
 - SCHEMA_ONLY
- Primary Keys
 - Must use for SCHEMA_AND_DATA durability
 - Specify inline or after columns
 - Must specify NONCLUSTERED

You can also specify a durability of SCHEMA_ONLY so that only the table definition is persisted. Any data in the table will be lost in the event of the database server shutting down. The ability to set the durability option to SCHEMA_ONLY is useful when the table is used for transient data, such as a session state table in a web server farm.

Primary Keys

All tables that have a durability option of SCHEMA_AND_DATA must include a primary key. You can specify this inline for single-column primary keys, or you can specify it after all of the column definitions. Memory-optimized tables do not support clustered primary keys. You must specify the word NONCLUSTERED when declaring the primary key.

To create a memory-optimized table, execute a CREATE TABLE statement that has the MEMORY_OPTIMIZED option set to ON, as shown in the following example:

Creating a Memory-Optimized Table

```
CREATE TABLE dbo.MemoryTable
(OrderId INTEGER NOT NULL PRIMARY KEY NONCLUSTERED,
 OrderDate DATETIME NOT NULL,
 ProductCode INTEGER NULL,
 Quantity INTEGER NULL)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

To create a memory-optimized table that has a composite primary key, you must specify the PRIMARY KEY constraint after the column definitions, as shown in the following example:

Creating a Memory-Optimized Table That Has a Composite Primary Key

```
CREATE TABLE dbo.MemoryTable2
(OrderId INTEGER NOT NULL,
 LineItem INTEGER NOT NULL,
 OrderDate DATETIME NOT NULL,
 ProductCode INTEGER NULL,
 Quantity INTEGER NOT NULL
PRIMARY KEY NONCLUSTERED (OrderId, LineItem))
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

Indexes in Memory-Optimized Tables

Memory-optimized tables support three kinds of indexes: Nonclustered hash indexes, nonclustered indexes, and columnstore indexes.

The most relevant use of columnstore indexes on memory-optimized tables is in real-time operational analytical processing, which is beyond the scope of this module.

For more information about columnstore indexes in real-time operational analytical processing see the topic *Get Started with Columnstore for Real-time Operational Analytics* in Microsoft Docs:

 **Get started with Columnstore for real time operational analytics**

<http://aka.ms/n7pu2j>

- Nonclustered hash indexes
 - Assign rows to buckets based on a hashing algorithm
 - Optimized for equality seeks
- Nonclustered indexes
 - Use a latch-free BW-tree structure
 - Support equality seeks, inequality seeks, and sort order
- Decide which type of index to use

When you create a memory-optimized table that has a primary key, an index will be created for the primary key. You can create up to seven other indexes in addition to the primary key. All memory-optimized tables must include at least one index, which can be the index that was created for the primary key.

Nonclustered Hash Indexes

Nonclustered hash indexes (also known simply as hash indexes) are optimized for equality seeks, but do not support range scans or ordered scans. Any queries that contain inequality operators such as `<`, `>`, and **BETWEEN** would not benefit from a hash index.

When you create a hash index for a primary key, or in addition to a primary key index, you must specify the bucket count. Buckets are storage locations in which rows are stored. You apply an algorithm to the indexed key values to determine the bucket in which the row is stored. When a bucket contains multiple rows, a linked list is created by adding a pointer in the first row to the second row, then in the second row to the third row, and so on.

To create a hash index, you must specify the `BUCKET_COUNT` value, as shown in the following example:

Creating a Table-Level Primary Key Hash Index in a Memory-Optimized Table

```
CREATE TABLE dbo.MemoryTable3
(OrderId INTEGER NOT NULL,
 LineItem INTEGER NOT NULL,
 ProductCode INTEGER NOT NULL
PRIMARY KEY NONCLUSTERED HASH (OrderId, LineItem) WITH (BUCKET_COUNT = 1000000)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

 **Note:** If there is more than one column in the key of the hash index, the `WHERE` clause of any query that uses the index must include equality tests for all columns in the key. Otherwise, the query plan will have to scan the whole table. When querying the table that was created in the preceding example, the `WHERE` clause should include an equality test for the **OrderId** and **LineItem** column.

You can create hash indexes, in addition to the primary key, by specifying the indexes after the column definitions, as shown in the following example:

Creating a Hash Index in Addition to the Primary Key in a Memory-Optimized Table

```
CREATE TABLE dbo.IndexedMemoryTable
(OrderId INTEGER NOT NULL,
 LineItem INTEGER NOT NULL,
 OrderDate DATETIME NOT NULL,
 ProductCode INTEGER NULL,
 Quantity INTEGER NOT NULL
PRIMARY KEY NONCLUSTERED HASH (OrderId, LineItem) WITH (BUCKET_COUNT = 1000000),
INDEX idx_MemTab_ProductCode NONCLUSTERED HASH(ProductCode) WITH (BUCKET_COUNT = 1000000)
)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

Nonclustered Indexes

Nonclustered indexes (also known as range indexes) use a latch-free variation of a binary tree (b-tree) structure, called a “BW-tree,” to organize the rows based on key values. Nonclustered indexes support equality seeks, range scans, and ordered scans.

You can create nonclustered indexes, in addition to the primary key, by specifying the indexes after the column definitions, as shown in the following example:

Creating a Nonclustered Index in Addition to the Primary Key in a Memory-Optimized Table

```
CREATE TABLE dbo.IndexedMemoryTable2
(OrderId INTEGER NOT NULL PRIMARY KEY NONCLUSTERED,
 OrderDate DATETIME NOT NULL,
 ProductCode INTEGER NULL,
 Quantity INTEGER NULL
INDEX idx_MemTab_ProductCode2 NONCLUSTERED (ProductCode)
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```



Note: SQL Server 2017 has improved the performance of nonclustered indexes on memory-optimized tables, thereby reducing the time required to recover a database.

Deciding Which Type of Index to Use

Nonclustered indexes benefit from a wider range of query operations. You should use a nonclustered index if any of the following scenarios apply:

- You might need to query the column or columns in the index key, using predicates with inequality operators such as <,>, or **BETWEEN**.
- The index key contains more than one column, and you might use queries with predicates that do not apply to all of the columns.
- The index requires a sort order.

If you are sure that none of the above scenarios apply, you could consider using a hash index to optimize equality seeks.



Note: You can have hash indexes and nonclustered indexes in the same table. Prior to SQL Server 2017, there was a limit of eight indexes including the primary key for memory-optimized tables. This limitation has been removed.

Converting Tables with Memory Optimization Advisor

Memory Optimization Advisor will review your existing disk-based tables and run through a checklist to verify whether your environment and the specific tables are suitable for you to convert the tables to memory-optimized tables.

Memory Optimization Advisor takes you through the following steps:

1. Memory Optimization Checklist

This step reports on any features of your disk-based tables that are not supported in memory-optimized tables.

- Memory Optimization Checklist
- Memory Optimization Warnings
- Review Optimization Options
- Review Primary Key Conversion
- Review Index Conversion
- Verify Migration Actions

2. Memory Optimization Warnings

Memory optimization warnings do not prevent a disk-based table from being migrated to a memory-optimized table, or stop the table from functioning after it has been converted—but the warnings will list any other associated objects, such as stored procedures, that might not function correctly after migration.

3. Review Optimization Options

You can now specify options such as the filegroup; the new name for the original, unmigrated, disk-based table; and whether to transfer the data from the original table to the new memory-optimized table.

4. Review Primary Key Conversion

If you are migrating to a durable table, you must specify a primary key or create a new primary key at this stage. You can also specify whether the index should be a hash index or not.

5. Review Index Conversion

This step gives you the same options as primary key migration for each of the indexes on the table.

6. Verify Migration Actions

This step lists the options that you have specified in the previous stages and enables you to migrate the table, or to create a script to migrate the table at a subsequent time.

To start Memory Optimization Advisor, in SQL Server Management Studio, right-click a table in Object Explorer, and then select **Memory Optimization Advisor**.

 **Note:** The Memory Optimization Advisor steps depend on the table. Actual pages may vary from those described above.

Querying Memory-Optimized Tables

When a database contains memory-optimized tables, there are two methods by which the tables can be queried. These methods involve using interpreted Transact-SQL and using natively compiled stored procedures.

Interpreted Transact-SQL

Transact-SQL in queries and stored procedures (other than native stored procedures) is referred to as interpreted Transact-SQL. You can use interpreted Transact-SQL statements to access memory-optimized tables in the same way as traditional disk-based tables. The SQL Server query engine provides an interop layer that does the necessary interpretation to query the compiled in-memory table. You can use this technique to create queries that access both memory-optimized tables and disk-based tables—for example, by using a JOIN clause. When you access memory-optimized tables, you can use most of the Transact-SQL operations that you use when accessing disk-based tables.

- Interpreted Transact-SQL
 - Enables memory-optimized tables in the same way as disk-based tables
 - Provides an interop layer
 - Enables queries that combine memory-optimized and disk-based tables
- Natively Compiled Stored Procedures
 - Increased performance
 - Stored procedure converted to C and compiled
 - Access to memory-optimized tables only

For more information about the Transact-SQL operations that are not possible when you access memory-optimized tables, see the topic *Accessing Memory-Optimized Tables Using Interpreted Transact-SQL* in Microsoft Docs:

Accessing Memory-Optimized Tables Using Interpreted Transact-SQL

<http://aka.ms/wycxxr>

Natively Compiled Stored Procedures

You can increase the performance of workloads that use memory-optimized tables further by creating natively compiled stored procedures. You can define these by using CREATE PROCEDURE statements that the SQL Server query engine converts to native C code. The C version of the stored procedure is compiled into a DLL, which is loaded into memory. You can only use natively compiled stored procedures to access memory-optimized tables; they cannot reference disk-based tables. You will see how to use natively compiled stored procedures in the next lesson.

Demonstration: Using Memory-Optimized Tables

In this demonstration, you will see how to:

- Create a database with a filegroup for memory-optimized data.
- Use memory-optimized tables.

Demonstration Steps

Create a Database with a Filegroup for Memory-Optimized Data

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**
2. In the **D:\Demofiles\Mod12** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**.
4. Start SQL Server Management Studio, and then connect to the **MIA-SQL** database engine instance by using Windows authentication.
5. In Object Explorer, under **MIA-SQL**, right-click **Databases**, and then click **New Database**.
6. In the **New Database** dialog box, in the **Database name** box, type **MemDemo**.
7. On the **Filegroups** page, in the **MEMORY OPTIMIZED DATA** section, click **Add Filegroup**.
8. In the **Name** box, type **MemFG**. Note that the filegroups in this section are used to contain FILESTREAM files because memory-optimized tables are persisted as streams.
9. On the **General** page, click **Add** to add a database file. Then add a new file that has the following properties:
 - **Logical Name:** MemData
 - **File Type:** FILESTREAM Data
 - **Filegroup:** MemFG
10. In the **Script** drop-down list, click **Script Action to New Query Window**.
11. In the **New Database** dialog box, click **Cancel** to view the script file that has been generated.

12. Review the script, noting the syntax that has been used to create a filegroup for memory-optimized data. You can use similar syntax to add a filegroup to an existing database.
13. Click **Execute** to create the database.

Use Memory-Optimized Tables

1. On the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the Open Project dialog box, navigate to **D:\Demofiles\Mod12\Demo12.ssmssln**, and then click **Open**.
3. In Solution Explorer, expand **Queries**, and then double-click **11 - Demonstration 1A.sql**.
4. Select the code under **Step 1 - Create a memory-optimized table**, and then click **Execute**.
5. Select the code under **Step 2 - Create a disk-based table**, and then click **Execute**.
6. Select the code under **Step 3 - Insert 500,000 rows into DiskTable**, and then click **Execute**.
This code uses a transaction to insert rows into the disk-based table.
7. When code execution is complete, look at the lower right of the query editor status bar, and note how long it has taken.
8. Select the code under **Step 4 - Verify DiskTable contents**, and then click **Execute**.
9. Confirm that the table now contains 500,000 rows.
10. Select the code under **Step 5 - Insert 500,000 rows into MemoryTable**, and then click **Execute**.
This code uses a transaction to insert rows into the memory-optimized table.
11. When code execution is complete, look at the lower right of the query editor status bar and note how long it has taken. It should be significantly lower than the time that it takes to insert data into the disk-based table.
12. Select the code under **Step 6 - Verify MemoryTable contents**, and then click **Execute**.
13. Confirm that the table now contains 500,000 rows.
14. Select the code under **Step 7 - Delete rows from DiskTable**, and then click **Execute**.
15. Note how long it has taken for this code to execute.
16. Select the code under **Step 8 - Delete rows from MemoryTable**, and then click **Execute**.
17. Note how long it has taken for this code to execute. It should be significantly lower than the time that it takes to delete rows from the disk-based table.
18. Select the code under **Step 9 - View memory-optimized table stats**, and then click **Execute**.
19. Close SQL Server Management Studio, without saving any changes.

Question: You are creating an index for a date column in a memory-optimized table. What is likely to be the most suitable type of index? Explain your reasons.

Lesson 2

Natively Compiled Stored Procedures

Natively compiled stored procedures are stored procedures that are compiled into native code. They are written in traditional Transact-SQL code, but are compiled when they are created rather than when they are executed, which improves performance.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the key features of natively compiled stored procedures.
- Create natively compiled stored procedures.

What Are Natively Compiled Stored Procedures?

Natively compiled stored procedures are written in Transact-SQL, but are then compiled into native code when they are created. This differs from traditional disk-based stored procedures (also known as interpreted stored procedures), which are compiled for the first time that they run. Compiling at creation time can cause errors that would not appear in an interpreted stored procedure until it is executed.

Natively compiled stored procedures access memory-optimized tables with greater speed and efficiency than interpreted stored procedures.

Natively compiled stored procedures contain one block of Transact-SQL that is called an atomic block. This block will either succeed or fail as a single unit. If a statement within a natively compiled stored procedure fails, the entire block will be rolled back to what it was before the procedure was executed. As you will see in the next topic, atomic blocks have three possible transaction isolation levels, which you must specify when creating a native stored procedure. Atomic blocks are not available to interpreted stored procedures.

For more information about natively compiled stored procedures, see the topic *Natively Compiled Stored Procedures* in Microsoft Docs:



Natively Compiled Stored Procedures

<http://aka.ms/lzfmaq>

- Written in Transact-SQL and compiled to native code at creation time
- Access memory-optimized tables
- Offer greater speed and efficiency
- Contain one atomic block—will succeed or fail as a single unit

When to use Natively Compiled Stored Procedures

Natively compiled stored procedures can give significant performance benefits, when used in the right situations. For best results, consider using natively compiled stored procedures when:

- Performance is critical. Natively compiled stored procedures work best in parts of an application that require fast processing.
- There are large numbers of rows to be processed. Natively compiled stored procedures give less benefit for single rowsets or when a small number of rows are returned.
- The stored procedure is called frequently. Because it is precompiled, you get a significant benefit for frequently used stored procedures.
- Logic requires aggregation functions, nested-loop joins, multistatement selects, inserts, updates, and deletes, or other complex expressions. They also work well with procedural logic—for example, conditional statements and loop constructs.

- Use natively compiled stored procedures when:
 - Performance is critical
 - There are many rows to be processed
 - The stored procedure is called frequently
 - Logic that uses:
 - Aggregation
 - Nested-loop joins
 - Multistatement select, insert, update, and deletes
 - Complex expressions
 - Procedural logic
 - Avoid named parameters

Finally, for best results, do not use named parameters with natively compiled stored procedures. Instead use ordinal parameters where the parameters are referred to by position.



Note: Natively compiled stored procedures only work with in-memory tables.

For more information about when to use a natively compiled stored procedure, see Microsoft Docs:



Best Practices for Calling Natively Compiled Stored Procedures

<https://aka.ms/alpha9>

Creating Natively Compiled Stored Procedures

To create a natively compiled stored procedure, you must use the CREATE PROCEDURE statement with the following options:

- NATIVE_COMPILATION
- SCHEMABINDING
- EXECUTE AS

In addition to these options, you must initiate a transaction in your stored procedure by using the BEGIN ATOMIC clause, specifying the transaction isolation level and language. You can specify one of the following transaction isolation levels:

- CREATE PROCEDURE statement
- NATIVE_COMPILATION option
- SCHEMABINDING option
- EXECUTE AS option
- BEGIN ATOMIC clause

- **SNAPSHOT.** Using this isolation level, all data that the transaction reads is consistent with the version that was stored at the start of the transaction. Data modifications that other, concurrent transactions have made are not visible, and attempts to modify rows that other transactions have modified result in an error.
- **REPEATABLE READ.** Using this isolation level, every read is repeatable until the end of the transaction. If another, concurrent transaction has modified a row that the transaction had read, the transaction will fail to commit due to a repeatable read validation error.
- **SERIALIZABLE.** Using this isolation level, all data is consistent with the version that was stored at the start of the transaction, and repeatable reads are validated. In addition, the insertion of "phantom" rows by other, concurrent transactions will cause the transaction to fail.

The following code example shows a CREATE PROCEDURE statement that is used to create a natively compiled stored procedure:

Creating a Natively Compiled Stored Procedure

```
CREATE PROCEDURE dbo.DeleteCustomer @CustomerID INT
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS
BEGIN ATOMIC WITH
    (TRANSACTION ISOLATION LEVEL = SNAPSHOT,
     LANGUAGE = 'us_English')
    DELETE dbo.Customer WHERE CustomerID = @CustomerID
    DELETE dbo.OpenOrders WHERE CustomerID = @CustomerID
END;
```

Some features are not supported in native stored procedures. For information about features that are not supported, see the "Natively Compiled Stored Procedures and User-Defined Functions" section in the topic *Transact-SQL Constructs Not Supported by In-Memory OLTP* in the SQL Server Technical Documentation:

Transact-SQL Constructs Not Supported by In-Memory OLTP

<https://aka.ms/l7lyio>

Execution Statistics

Execution statistics for natively compiled stored procedures is not enabled by default. There is a small performance impact with collecting statistics, so you must explicitly enable the option when you need it. You can enable or disable the collection of statistics using sys.sp_xtp_control_proc_exec_stats.

Once you have enabled the collection of statistics, you can monitor performance using sys.dm_exec_procedure_stats.

- To enable the collection of statistics, use sys.sp_xtp_control_proc_stats
- To monitor the performance of a natively compiled stored procedure, use sys.dm_exec_procedure_stats

Use a dynamic management view (DMV) to collect statistics, after you have enabled the collection of statistics.

Sys.dm_exec_procedure_stats

```
SELECT *
FROM sys.dm_exec_procedure_stats
```



Note: Statistics are not automatically updated for in-memory tables. You must use the **UPDATE STATISTICS** command to update specific tables or indexes, or the **sp_updatestats** to update all the statistics.

Demonstration: Creating a Natively Compiled Stored Procedure

In this demonstration, you will see how to:

- Create a natively compiled stored procedure.

Demonstration Steps

Create a Natively Compiled Stored Procedure

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**
2. Ensure that you have run the previous demonstration.
3. Start SQL Server Management Studio, and then connect to the **MIA-SQL** database engine instance by using Windows authentication.
4. On the **File** menu, click **Open**, click **Project/Solution**.
5. In the **Open Project** dialog box, navigate to **D:\Demofiles\Mod12\Demo12.ssmssln**, and then click **Open**.
6. In Solution Explorer, expand **Queries**, and then double-click **21 - Demonstration 2A.sql**.
7. Select the code under **Step 1 - Use the MemDemo database**, and then click **Execute**.
8. Select the code under **Step 2 - Create a native stored proc**, and then click **Execute**.
9. Select the code under **Step 3 - Use the native stored proc**, and then click **Execute**.
10. Note how long it has taken for the stored procedure to execute. This should be significantly lower than the time that it takes to insert data into the memory-optimized table by using a Transact-SQL **INSERT** statement.
11. Select the code under **Step 4 - Verify MemoryTable contents**, and then click **Execute**.
12. Confirm that the table now contains 500,000 rows.
13. Close SQL Server Management Studio without saving any changes.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
<p>You are executing a native stored procedure that inserts a row into a memory-optimized table for customer data, and then inserts a row into a memory-optimized table for sales data. The row is inserted into the customer table successfully, but the statement to insert the row into the sales table fails, causing the procedure to return an error.</p> <p>True or false? When you check the tables by running a SELECT query, the row that was successfully inserted will show in the customer table.</p>	

Lab: Using In-Memory Database Capabilities

Scenario

You are planning to optimize some database workloads by using the in-memory database capabilities of SQL Server. You will create memory-optimized tables and natively compiled stored procedures to optimize OLTP workloads.

Objectives

After completing this lab, you will be able to:

- Create a memory-optimized table.
- Create a natively compiled stored procedure.

Estimated Time: 45 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Using Memory-Optimized Tables

Scenario

The Adventure Works website, through which customers can order goods, uses the InternetSales database. The database already includes tables for sales transactions, customers, and payment types. You need to add a table to support shopping cart functionality. The shopping cart table will experience a high volume of concurrent transactions, so, to maximize performance, you want to implement it as a memory-optimized table.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Add a Filegroup for Memory-Optimized Data
3. Create a Memory-Optimized Table

► Task 1: Prepare the Lab Environment

1. Ensure that the MIA-DC and MIA-SQL virtual machines are both running, and then log on to MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab12\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. When you are prompted, click **Yes** to confirm that you want to run the command file, and then wait for the script to finish.

► Task 2: Add a Filegroup for Memory-Optimized Data

1. Add a filegroup for memory-optimized data to the InternetSales database.
2. Add a file for memory-optimized data to the InternetSales database. You should store the file in the filegroup that you created in the previous step.

► **Task 3: Create a Memory-Optimized Table**

1. Create a memory-optimized table named **ShoppingCart** in the InternetSales database, with the durability option set to SCHEMA_AND_DATA.
2. The table should include the following columns:
 - o **SessionID**: integer
 - o **TimeAdded**: datetime
 - o **CustomerKey**: integer
 - o **ProductKey**: integer
 - o **Quantity**: integer
3. The table should include a composite primary key nonclustered index on the **SessionID** and **ProductKey** columns.
4. To test the table, insert the following rows, and then write and execute a SELECT statement to return all of the rows.

SessionID	TimeAdded	CustomerKey	ProductKey	Quantity
1	<Time>	2	3	1
1	<Time>	2	4	1

For <Time>, use whatever the current time is.

Results: After completing this exercise, you should have created a memory-optimized table and a natively compiled stored procedure in a database with a filegroup for memory-optimized data.

Exercise 2: Using Natively Compiled Stored Procedures

Scenario

The Adventure Works website now includes a memory-optimized table. You want to create a natively compiled stored procedure to take full advantage of the performance benefits of in-memory tables.

The main tasks for this exercise are as follows:

1. Create Natively Compiled Stored Procedures

► **Task 1: Create Natively Compiled Stored Procedures**

1. Create a natively compiled stored procedure named **AddItemToCart**. The stored procedure should include a parameter for each column in the **ShoppingCart** table, and should insert a row into the **ShoppingCart** table by using a SNAPSHOT isolation transaction.
2. Create a natively compiled stored procedure named **DeleteItemFromCart**. The stored procedure should include **SessionID** and **ProductKey** parameters, and should delete matching rows from the **ShoppingCart** table by using a SNAPSHOT isolation transaction.
3. Create a natively compiled stored procedure named **EmptyCart**. The stored procedure should include **SessionID** parameters, and should delete matching rows from the **ShoppingCart** table by using a SNAPSHOT isolation transaction.

4. To test the **AddItemToCart** procedure, write and execute a Transact-SQL statement that calls **AddItemToCart** to add the following items, and then write and execute a SELECT statement to return all of the rows in the **ShoppingCart** table.

SessionID	TimeAdded	CustomerKey	ProductKey	Quantity
1	<Time>	2	3	1
1	<Time>	2	4	1
3	<Time>	2	3	1
3	<Time>	2	4	1

For *<Time>*, use whatever the current time is.

5. To test the **DeleteItemFromCart** procedure, write and execute a Transact-SQL statement that calls **DeleteItemFromCart** to delete any items where **SessionID** is equal to 3 and the product key is equal to 4, and then write and execute a SELECT statement to return all of the rows in the **ShoppingCart** table.
6. To test the **EmptyCart** procedure, write and execute a Transact-SQL statement that calls **EmptyCart** to delete any items where **SessionID** is equal to 3, and then write and execute a SELECT statement to return all of the rows in the **ShoppingCart** table.
7. Close SQL Server Management Studio without saving any changes.

Results: After completing this exercise, you should have created a natively compiled stored procedure.

Module Review and Takeaways

In this module, you have learned how to:

- Use memory-optimized tables to improve performance for latch-bound workloads.
- Use natively compiled stored procedures.

Review Question(s)

Check Your Knowledge

Question
Which of the following statements is true?
Select the correct answer.
<input type="checkbox"/> Interpreted stored procedures cannot be applied to memory-optimized tables.
<input type="checkbox"/> Native stored procedures cannot be applied to disk-based tables.
<input type="checkbox"/> Native stored procedures can be applied to memory-optimized tables and to disk-based tables.
<input type="checkbox"/> Interpreted stored procedures can contain atomic blocks.
<input type="checkbox"/> Native stored procedures compile the code the first time the stored procedure is executed.

Module 13

Implementing Managed Code in SQL Server

Contents:

Module Overview	13-1
Lesson 1: Introduction to CLR Integration in SQL Server	13-2
Lesson 2: Implementing and Publishing CLR Assemblies	13-9
Lab: Implementing Managed Code in SQL Server	13-17
Module Review and Takeaways	13-20

Module Overview

As a SQL Server® professional, you are likely to be asked to create databases that meet business needs. Most requirements can be met using Transact-SQL. However, occasionally you may need additional capabilities that can only be met by using common language runtime (CLR) code.

As functionality is added to SQL Server with each new release, the necessity to use managed code decreases. However, there are times when you might need to create aggregates, stored procedures, triggers, user-defined functions, or user-defined types. You can use any .NET Framework language to develop these objects.

In this module, you will learn how to use CLR managed code to create user-defined database objects for SQL Server.

Objectives

After completing this module, you will be able to:

- Explain the importance of CLR integration in SQL Server.
- Implement and publish CLR assemblies using SQL Server Data Tools (SSDT).

Lesson 1

Introduction to CLR Integration in SQL Server

Occasionally, you might want to extend the built-in functionality of SQL Server; for example, adding a new aggregate to the existing list of aggregates supplied by SQL Server.

CLR integration is one method for extending SQL Server functionality. This lesson introduces CLR integration in SQL Server, and its appropriate use cases.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the ways in which you can extend SQL Server.
- Describe the .NET Framework.
- Describe the .NET CLR environment.
- Explain the need for managed code in SQL Server.
- Explain the situations when the use of Transact-SQL is inappropriate.
- Choose appropriate use cases for managed code in SQL Server.

Options for Extending SQL Server Functionality

With Microsoft® you can extend SQL Server's functionality in several ways:

CLR Managed Code

Managed code is normally written in Microsoft Visual C#® or Microsoft Visual Basic®, and is executed under the management of the .NET Framework CLR. Managed code is safer than unmanaged code—it is designed to be more reliable and robust. You can extend the functionality of SQL Server by using managed code to create user-defined types, aggregates, mathematical functions, and other functionality.

- With CLR managed code, you can extend the functionality of SQL Server
 - Executes under the management of the .NET Framework CLR
 - Written in Visual C# or Visual Basic
 - Use it to create user-defined types, aggregates, mathematical functions, and other functionality
 - SQL Server components such as SSIS, SSAS, and SSRS are also extensible
 - Extended stored procedures are deprecated
 - Prone to memory leaks and other performance issues
 - Use CLR managed code instead

SQL Server Components

In addition to the database engine, SQL Server now includes Analysis Services (SSAS), Reporting Services (SSRS), and Integration Services (SSIS). These components are also extensible. For example, you use SSRS to create rendering extensions, security extensions, data processing extensions, delivery extensions, custom code, and external assemblies.

Extended Stored Procedures

Previous versions of SQL Server used extended stored procedures to extend the database's functionality. These were written in C++ and executed directly within the address space of the SQL Server engine. This led to memory leaks and other performance issues because minor errors could cause instabilities. Microsoft does not recommend the use of extended stored procedures, and their use is effectively deprecated. CLR managed code should be used in place of extended stored procedures.

This module focuses on using CLR managed code to extend SQL Server functionality.

Introduction to the .NET Framework

The .NET Framework is the foundation for developing Windows® applications and services, including extended functionality for SQL Server. The .NET Framework provides development tools that make application and service development easier.

Win32 and Win64 APIs

The Windows operating system has evolved over many years. Win32 and Win64 application programming interfaces (APIs) are the programming interfaces to the operating system. These interfaces are complex and often inconsistent because they have evolved over time rather than being designed with a single set of guidelines.

- The Win32 and Win64 APIs evolved over time
 - They are complex and inconsistent in their design
- The .NET Framework is an object-oriented development framework
 - It is consistent and well-designed
 - It includes thousands of class libraries
 - It provides a layer of abstraction above the Windows operating system
 - Generally well-regarded by developers
- .NET Framework provides a good basis for writing code to extend SQL Server functionality

.NET Framework

The .NET Framework is a layer of software that sits above the Win32 and Win64 APIs, and provides a layer of abstraction above the underlying complexity. The .NET Framework is object-oriented and written in a consistent fashion to a tightly defined set of design guidelines. Many people describe it as appearing to have been “written by one brain.” It is not specific to any one programming language, and contains thousands of prebuilt and pretested objects. These objects are collectively referred to as the .NET Framework class libraries.

The .NET Framework is generally well regarded amongst developers, making it a good choice for building code to extend SQL Server.

.NET Common Language Runtime

The .NET CLR is a layer in the .NET Framework which runs code and services that simplify development and deployment. With the CLR, you can create programs and procedures in any .NET language to produce managed code. Managed code executes under the management of the common language runtime virtual machine.

By using the CLR integration feature within SQL Server, you can use .NET assemblies to extend SQL

- CLR is the runtime environment for the .NET Framework—it provides a number of services including:
 - Running code
 - Providing services
 - Avoiding memory leaks through garbage collection
 - Destroys objects that are no longer used
 - Operating within other programming environments
 - Enabling interoperability between languages through the common language specification (CLS)

Server functionality. The .NET CLR offers:

- Better memory management.
- Access to existing managed code.
- Security features to ensure that managed code will not compromise the server.
- The ability to create new resources by using .NET Framework languages such as Microsoft Visual C# and Microsoft Visual Basic .NET.

Memory Management

Managing memory allocation was a problem when developing directly to the Win32 and Win64 APIs. Component Object Model (COM) programming preceded the .NET Framework, using reference counting to release memory that was no longer needed. COM would work something like this:

1. Object A creates object B.
2. When Object B is created, it notes that it has one reference.
3. Object C acquires a reference to object B. Object B then notes that it has two references.
4. Object C releases its reference. Object B then notes that it has one reference.
5. Object A releases its reference to Object B. Object B then notes that it has no references, so it is destroyed.

The problem was that it was easy to create situations where memory could be lost. Consider a circular reference: if two objects have references to each other, with no other references to either of them, they can consume memory providing they have a reference to each other. This causes a leak, or loss, of the memory. Over time, this badly written code results in a loss of all available memory, resulting in instability and crashes. This is obviously highly undesirable when integrating code into SQL Server.

The .NET Framework includes a sophisticated memory management system, which is known as garbage collection, to avoid memory leaks. There is no referencing counting—instead the CLR periodically checks which objects are “reachable” and disposes of the other objects.

Type Safety

Type safe code accesses memory in a properly structured way.

Type safety is a problem with Win32 and Win64 code. When a function or procedure is called, all that is known to the caller is the function’s address in memory. The caller assembles a list of required parameters, places them in an area called the stack, and jumps to the memory address of the function. Problems can arise when the design of the function changes, but the calling code is not updated. The calling code can then refer to memory locations that do not exist.

The .NET CLR is designed to avoid such problems. Objects are isolated from one another and can only access the memory allocations for which they have permissions. In addition to providing address details of a function, the CLR also provides the function’s signature. The signature specifies the data types of each of the parameters, and their order. The CLR will not permit a function to be called with the wrong number or types of parameters.

Hosting the CLR

The CLR is designed to be hostable, meaning that it can be operated from within other programming environments, such as SQL Server. SQL Server acts as the operating system, controlling CLR memory management, stability, and performance. For example, SQL Server manages the memory that a CLR assembly can address so that code cannot affect database engine processes.

For more information about the CLR hosted environment within SQL Server, see Microsoft Docs:



CLR Hosted Environment

<http://aka.ms/K9pk7y>

Common Language Specification

The common language specification (CLS) specifies the rules to which languages must conform. A simple example is that C# is case sensitive, and can recognize two methods, named `SayHello` and `Sayhello`, as being distinct from each other. These two methods could not be called from a case-insensitive language because they would appear to be the same. The CLS avoids interoperability problems like this by not permitting you to name these two methods with the same name, regardless of case.

Why Use Managed Code with SQL Server?

Transact-SQL is the primary tool to work with relational data in SQL Server; it works efficiently with the SQL Server Database Engine, and covers most eventualities. So it is relatively rare that managed code is required. Occasionally, however, you might have to do something that is outside the scope of Transact-SQL.

.NET Framework Classes

The .NET Framework offers a set of libraries, each of which contains a large set of prewritten and pretested objects—typically referred to as classes.

For example, the Regular Expression (RegEx) library is a powerful string manipulation class that you can use within SQL Server by using the CLR integration feature.

The inclusion of managed code in SQL Server also makes access to external resources easier, and in some cases, provides higher performance.

- The .NET Framework provides a rich class library
- Managed code can be used to create objects that you normally create using Transact-SQL:
 - User-defined functions (scalar and table-valued)
 - Stored procedures
 - Triggers (DML and DDL)
- Managed code can be used to create new types of objects:
 - User-defined data types
 - User-defined aggregates

Alternative to Transact-SQL Objects

Many objects that you can create in Transact-SQL can also be created in managed code, including:

- Scalar user-defined functions.
- Table-valued user-defined functions.
- Stored procedures.
- Data manipulation language (DML) triggers.
- Data definition language (DDL) triggers.

New Object Types

In managed code, you can also construct types of objects that you cannot construct in Transact-SQL. These include the following:

- User-defined data types.
- User-defined aggregates.

Although you can create objects using managed code, it does not necessarily mean that you should. Transact-SQL should be used most of the time, with managed code used only when necessary.

Considerations When Using Managed Code

When you are considering whether to use Transact-SQL or managed code, there are a number of considerations.

Portability

Upgrading a database system that includes managed code can be more complicated. From time to time, SQL Server must be upgraded when old versions come to the end of their life, and managed code may or may not work with newer versions, depending on functionality. This can also be an issue with Transact-SQL, but Transact-SQL is more likely to have an upgrade path. CLR managed code is also dependent on the .NET Framework version installed on the server.

- Considerations when using managed code:
 - Portability—upgrading your database
 - Maintainability—consider expertise required to maintain:
 - The database
 - Managed code components
 - Consider using a three-tier architecture
 - Database tier
 - Mid tier
 - Presentation tier
 - Transact-SQL is well suited to working with SQL Server tables and data
 - Use managed code sparingly; otherwise consider a three-tier architecture

Maintainability

Database administrators (DBAs) generally have a good knowledge of Transact-SQL, but little or no knowledge of C# or Visual Basic. Adding managed code to a database system means that additional expertise may be required to maintain the system. For larger organizations that already employ developers, this may not be a problem. Organizations that rely on DBAs to support their SQL Server databases may find that adding managed code creates a split in expertise that, over time, causes problems.

Three-Tier Architecture

Transact-SQL is designed as an efficient language to work with relational database tables. If you have an extensive need for managed code, consider the three-tier architecture for your system. Each tier is constructed separately, possibly by different teams with different skills. There is a boundary between each tier, so that each one can be properly and independently tested. Each tier is built using the development tools best suited to its needs. This separation of concerns creates systems that are more maintainable, and faster to develop.

A typical three-tier architecture might be composed of a:

- **Database tier.** The tables, views, stored procedures and other database objects.
- **Mid tier or business tier.** The data access objects and other code that manage the business logic. As the name suggests, the mid tier (or business tier) sits between the database tier and the presentation tier.
- **Presentation tier.** This is the user interface tier, which might include forms for data input, reports, and other content.

Transact-SQL

Transact-SQL is the primary method for manipulating data within databases. It is designed for direct data access and has many built-in functions. However, Transact-SQL is not a fully-fledged high level programming language. It is not object-oriented so, for example, you cannot create a stored procedure that takes a parameter of an animal data type and pass a parameter of a cat data type to it. Also, Transact-SQL is not designed for tasks such as intensive calculations or string handling.

Managed Code

Managed code provides full object-oriented capabilities, although this only applies within the managed code itself. Managed code works well within SQL Server when used sparingly; otherwise you should consider using a mid tier.

General Rules

Two good general rules apply when you are choosing between using Transact-SQL and managed code:

- Data-oriented requirements should almost always be handled using Transact-SQL.
- Some specialist calculations, strings, or external access might require managed code.

Appropriate Use of Managed Code

In the last topic, we discussed some considerations when planning to use managed code within your database. The following list describes areas that might be suitable for managed code, if needed:

Scalar UDFs

Some scalar user-defined functions (UDFs) that are written in Transact-SQL cause performance problems. Managed code can provide an alternative way of implementing scalar UDFs, particularly when the function does not depend on data access.

Database object	Transact-SQL	Managed code
Scalar UDF	Usual implementation despite some performance issues	Candidate when there is no data access
Table-valued function	Usual implementation	Candidate option for non-data related situations
Stored procedure	Usual implementation	Rarely a good option
DML trigger	Almost always	Rarely a good option
DDL trigger	Usual implementation	Candidate option in some situations, for example XML processing of EVENTDATA
Aggregate	Not possible	Good option
User-defined data type	Aliased data types only	Good option

Table-Valued UDFs

Data-related table-valued UDFs are generally best implemented using Transact-SQL. However, table-valued UDFs that have to access external resources, such as the file system, environment variables, or the registry, might be candidates for managed code. Consider whether this functionality properly sits within the database layer, or whether it should be handled outside of SQL Server.

Stored Procedures

With few exceptions, stored procedures should be written in Transact-SQL. The exceptions to this are stored procedures that have to access external resources or perform complex calculations. However, you should consider whether code that performs these tasks should be implemented within SQL Server at all—it might be better implemented in a mid tier.

DML Triggers

Almost all DML triggers are heavily oriented toward data access and should be written in Transact-SQL. There are very few valid use cases for implementing DML triggers in managed code.

DDL Triggers

DDL triggers are also data-oriented. However, some DDL triggers have to do extensive XML processing, particularly based on the XML EVENTDATA structure that SQL Server passes to these triggers. The more that extensive XML processing is required, the more likely it is that the DDL trigger would be best implemented in managed code. Managed code would also be a better option if the DDL trigger needed to access external resources—but this is rarely a good idea within any form of trigger. Again, for any but the lightest use, consider implementing a mid tier.

User-Defined Aggregates

Transact-SQL has no concept of user-defined aggregates. You have to implement these in managed code.

User-Defined Data Types

Transact-SQL offers the ability to create alias data types, but these are not really new data types, they are subsets of existing built-in data types. Managed code offers the ability to create entirely new data types to determine what data needs to be stored, and the behavior of the data type.

Check Your Knowledge

Question	
Why might you include managed code in your SQL Server dataset?	
Select the correct answer.	
	To create a new data type that is used widely in your database.
	To replace some Transact-SQL code that is running slowly.
	To create a trigger that alters code in another application.
	To back up SQL Server at a certain time on Monday morning.

Lesson 2

Implementing and Publishing CLR Assemblies

There are two ways to deploy a CLR assembly to a computer running SQL Server—either with Transact-SQL scripts or using SQL Server Data Tools (SSDT). This lesson focuses on using SSDT to develop and deploy CLR assemblies. Code examples have been written using C#.

Lesson Objectives

After completing this lesson, you will:

- Be able to explain what an assembly is.
- Understand assembly permissions.
- Use SSDT to create a CLR assembly.
- Use SSDT to publish a CLR assembly.

What Is an Assembly?

Managed code is deployed in SQL Server within an assembly—a .dll file that contains the executable code and a manifest. The manifest describes the contents of the assembly, and the interfaces to the assembly. SQL Server and other code can then interrogate what the assembly contains and what it can do.

Assemblies can contain other resources such as icons, which are also listed in the manifest. In general terms, assemblies can be either .exe files or .dll files; however, SQL Server only works with .dll files.

- Managed code is deployed in SQL Server using an assembly
- A SQL Server assembly is:
 - A specially structured .dll file
 - Self-describing through its manifest
- Assemblies contain compiled executable managed code
 - They might also contain other resources
- Use SQL Server Data Tools (SSDT) to create managed code, and to publish it to SQL Server

Deployment and Security

Assemblies are created outside of SQL Server and they provide the functionality to deploy and version managed code. After creating an assembly, you can share it between SQL Server instances and business applications.

Security is applied at the assembly level.

In this lesson, you will see how to use SSDT to create assemblies and publish them to SQL Server.

Assembly Permission Sets

The CLR offers several levels of trust that you can set within policies for the host machine on which the assembly runs. There are three SQL Server permissions with which the administrator can control the server's exposure: SAFE, EXTERNAL_ACCESS, and UNSAFE.

Regardless of what the code in an assembly attempts to do, the permission set determines the permitted actions.

- Sp_configure has an option called CLR strict security
 - Improves the security of CLR assemblies
 - CLR Strict Security set to 1 by default
 - Can be set to 0 to disable, but not recommended
- CLR Strict Security overrides
 - PERMISSION_SET = SAFE
 - PERMISSION_SET = EXTERNAL_ACCESS
 - Both treated as UNSAFE
- Add trusted assemblies to a white list
 - Sys.sp_add_trusted_assembly

SAFE

SAFE assemblies have a limited permission set, and only provide access for the SQL Server database in which it is cataloged. SAFE is the default permission set—it's the most restrictive and secure. Assemblies with SAFE permissions cannot access the external system; for example, network files, the registry, or other files external to SQL Server.

EXTERNAL_ACCESS

EXTERNAL_ACCESS is the permission set that is required to access local and network resources such as environment variables and the registry. Assemblies with EXTERNAL_ACCESS permissions cannot be used within a contained database.

UNSAFE

UNSAFE is the unrestricted permission set that should rarely, if ever, be used in a production environment. UNSAFE is required for code that calls external unmanaged code, or code that holds state information across function calls. UNSAFE assemblies cannot be used in a contained database.

Setup for EXTERNAL_ACCESS and UNSAFE

The EXTERNAL_ACCESS and UNSAFE permission sets require a trust level to be set up before you can use them. There are two ways to do this:

- You can flag the database as TRUSTWORTHY by using the ALTER DATABASE SET TRUSTWORTHY ON statement. This is not recommended as, under certain circumstances, it could provide access for malicious assemblies.
- Create an asymmetric key from the assembly file that is cataloged in the **master** database—then create a login mapping to that key. Finally, grant the login EXTERNAL ACCESS ASSEMBLY permission on the assembly. This is the recommended method of granting permission to use the EXTERNAL_ACCESS or UNSAFE permission sets.

Setting Permissions

When you create an assembly using SSDT, the default permission set is SAFE. Alternatively, you can use the CREATE ASSEMBLY <Transact-SQL clause> WITH PERMISSION_SET = {SAFE | EXTERNAL_ACCESS | UNSAFE}.

To change the permission set using SSDT, use the **Properties** tab to set the **Permission** level. Right-click the assembly name, and then click **Properties**.

For more information about permissions, see Microsoft Docs:

Creating an Assembly

<http://aka.ms/ljs5b1>

SP_CONFIGURE

For security reasons, SQL Server does not allow CLR integration by default. To enable CLR integration, you must set the **clr enabled** option to **1**. This is set at the instance level using the **sp_configure** stored procedure.

This example code firstly displays the settings for **sp_configure**, enables the advanced options to be displayed, and then sets the **clr enabled** option to **1**. This allows CLR managed code to be run within the SQL Server instance:

Using sp_configure to Enable CLR Integration

```
sp_configure;
GO

sp_configure 'show advanced options', 1;
GO
RECONFIGURE;
GO

sp_configure 'clr enabled', 1;
GO
RECONFIGURE;
GO
```

CLR Strict Security

CLR Strict Security is a setting in **sp_configure** that overrides the settings of **SAFE** and **EXTERNAL ACCESS**. When CLR Strict Security is set to 1, the **PERMISSION_SET** information is ignored, and all assemblies are treated as **UNSAFE**.

CLR Strict Security is set to 1 by default, but can be disabled by setting it to 0 although this is not recommended. Instead, consider signing assemblies with a certificate or using an asymmetric key with a login that has **UNSAFE ASSEMBLY** permissions..

sys.sp_add_trusted_assembly

Trusted assemblies can be added to a "white list" using **sp_add_trusted_assembly**. Use **sp_add_trusted_assembly**, **sp_drop_trusted_assembly**, and **sys.trusted_assemblies** to manage your whitelist.

For more information about CLR strict security, and see Microsoft Docs:

CLR strict security

<https://aka.ms/C3kl7u>

For more information about **sys.sp_add_trusted_assembly**, see Microsoft Docs:

sys.sp_add_trusted_assembly (Transact-SQL)

<https://aka.ms/Jggb1s>

SQL Server Data Tools

SQL Server Data Tools (SSDT) was introduced with SQL Server 2012 to provide a rich development environment for SQL Server. SSDT is integrated into Visual Studio and so is familiar to many developers—although perhaps less familiar to DBAs. You can use SSDT to develop, debug, and refactor database code, in addition to developing Transact-SQL and CLR managed code.

CLR managed code must be created for the specific .NET Framework version that is deployed on the target machine. With SSDT, you can specify the .NET Framework version you are developing it for. SSDT also provides a number of templates for different SQL Server objects including:

- Aggregates
- Stored procedures
- Triggers
- User-defined functions
- User-defined types

- SQL Server Development Tools (SSDT) was introduced with SQL Server 2012
- Integrates with Visual Studio
 - Familiar development environment
 - Use SSDT to develop and deploy CLR assemblies
- SSDT templates for:
 - Aggregates
 - Stored procedures
 - Triggers
 - User-defined functions
 - User-defined types
- Permission level is **SAFE** by default

Develop a CLR Assembly Using SSDT

You can develop a CLR managed code assembly using Visual Studio 2015 with SSDT installed, using the following steps:

1. Determine which version of the .NET Framework is installed on the target SQL Server computer.
2. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.
3. In the list of templates, click **SQL Server** and then click **SQL Server Database Project**. Select the .NET Framework of your target SQL Server computer, enter a name and directory for your project and then click **OK**.
4. To create a CLR object, add a new item to the project using the template for the language of the SQL CLR object you want to create. The examples in this course are written in C#, so you would select **SQL CLR C#**.
5. Select the type of CLR object you want to create. The available choices include aggregates, stored procedures, triggers, user-defined functions, and data types.
6. In the new template code window, write your code.
7. When you have completed your CLR code, build the solution, correcting any errors that might occur.
8. Publish the CLR assembly, specifying the target database and connection information. You can then use the CLR assembly within your SQL Server database.

Set Permission Level

The permission level is set at the assembly level in the **Properties** dialog box for that assembly. You can set the permission level by selecting the appropriate option from the drop-down list. **SAFE** is the default option, and any assemblies you create will automatically be set to **SAFE** if you do not amend the permission level.

For a good introduction to SQL Server Data Tools, see MSDN:



SQL Server Data Tools

<http://aka.ms/Koihog>

Publishing a CLR Assembly

Visual Studio and SSDT is an ideal environment for developing CLR managed code and most C# developers will be familiar with Visual Studio.

SSDT includes a number of preinstalled code templates, including:

- SQL CLR C# aggregate.
- SQL CLR C# stored procedure.
- SQL CLR C# user-defined function.
- SQL CLR C# user-defined type.

The templates include generated code that you can customize with your own code. You then publish CLR managed code assemblies to on-premises SQL Server instances.

Set Properties for the Assembly

After creating the assembly, you have to configure its properties. You must set the permission level to SAFE, and the target platform must be the version of SQL Server where you are deploying your assembly. Properties and permissions are both set at the assembly level. The permission level default is SAFE and the default target platform is SQL Server 2016.

Using SSDT, open the CLR assembly project, and in Solution Explorer, right-click on the project name, and then click **Properties**. On the **Project Settings** page, amend the target platform. On the **SQLCLR** page, set the permission level. Save the assembly before closing the **Properties** dialog box.

Rebuild and Publish the Assembly

After configuring the properties, rebuild the solution, and then publish it to your target database.

Using the Assembly

After the assembly is published, you will be able to access it in SSMS—in Solution Explorer, expand your database, expand **Programmability**, and then drill down to the appropriate type of assembly. In addition, you can access it directly from your Transact-SQL code.

- Visual Studio and SSDT is a familiar environment for .NET software developers
- SSDT contains a number of SQL CLR templates:
 - Aggregate
 - Stored procedure
 - User-defined function
 - User-defined type
- Set properties for the assembly
 - Permission level
 - Target platform
- Build the assembly and publish to your database

Demonstration: Creating a User-Defined Function

In this demonstration, you will see how to:

- Develop a simple function using CLR C# managed code.
- Publish an assembly.

Demonstration Steps

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Navigate to **D:\Demofiles\Mod13**, 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. On the Start screen, type **SQL Server Data Tools 2015**, and then click **SQL Server Data Tools 2015**.
5. On the **File** menu, point to **New**, and then click **Project**.
6. In the **New Project** dialog box, expand **Templates**, and then click **SQL Server**.
7. In the middle pane, click **SQL Server Database Project**, in the top pane, in the **.NET Framework** list, click **.NET Framework 4.6**.
8. In the **Name** box, type **ClrDemo**, in the **Location** box, type **D:\Demofiles\Mod13**, and then click **OK**.
9. In Solution Explorer, right-click **ClrDemo**, point to **Add**, and then click **New Item**.
10. In the **Add New Item** dialog box, in the **Installed** list, under **SQL Server**, click **SQL CLR C#**, in the middle pane, click **SQL CLR C# User Defined Function**, in the **Name** box, type **HelloWorld.cs**, and then click **Add**.
11. Locate the **return** statement, which is immediately below the comment **// Put your code here**.
12. Amend the function to read:


```
return new SqlString("Hello World!");
```
13. On the **File** menu, click **Save All**.
14. In Solution Explorer, right-click **ClrDemo**, and then click **Properties**.
15. On the **Project Settings** page, in the **Target platform** list, select **SQL Server 2017**.
16. On the **SQLCLR** page, in the **Permission level** list, note that **SAFE** is selected, then click **Signing**.
17. In the **Signing** dialog, select **Sign the assembly**. In the **Choose a strong name key file** box, select **New**.
18. In the **Create Strong Name Key** dialog, in the **Key file name** box type **ClrDemo**. In the **Enter password** box type **Pa55w.rd** then in the **Confirm password** box type **Pa55w.rd** then click **OK**, then click **OK**.
19. On the **File** menu, click **Save All**.
20. On the **Build** menu, click **Build Solution**.
21. In Windows Explorer, navigate to **D:\Demofiles\Mod13**, then double-click **create_login.sql**. When SQL Server Management Studio starts, in the **Connect to Database Engine** dialog, confirm that the **Server name** box has the value **MIA-SQL**, then click **Connect**. If the **Connect to Database Engine** dialog is displayed a second time, click **Connect**.

22. Review the script in the `create_login.sql` pane. Observe that an asymmetric key is created from the `ClrDemo.dll` that you just compiled, and that a login is created based on the imported asymmetric key, then click **Execute**. When the script completes, close SQL Server Management Studio.
23. In Solution Explorer, right-click **ClrDemo**, and then click **Publish**.
24. In the **Publish Database** dialog box, click **Edit**.
25. In the **Connect** dialog box, on the **Browse** tab, expand **Local**, and then click **MIA-SQL**.
26. In the **Database Name** list, click **AdventureWorks2014**, and then click **Test Connection**.
27. In the **Connect** message box, click **OK**.
28. In the **Connect** dialog box, click **OK**.
29. In the **Publish Database** dialog box, click **Publish**.
30. After a few seconds, a message will be displayed to say it has published successfully.
31. On the taskbar, click **Microsoft SQL Server Management Studio**.
32. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, and then click **Connect**.
33. In Object Explorer, expand **Databases**, expand **AdventureWorks2014**, expand **Programmability**, expand **Functions**, and then expand **Scalar-valued Functions**. Note that **dbo.HelloWorld** appears in the list.
34. In Object Explorer, right-click **AdventureWorks2014**, and click **New Query**.
35. In the new query window, type the following code, and then click **Execute**.

```
SELECT dbo.HelloWorld();
```

36. Close SSMS without saving any changes.
37. Close Visual Studio.

Question: Do you use managed code in your SQL Server databases? Who maintains the code when it needs amending?

Sequencing Activity

Put the following steps in order by numbering each to indicate the correct order:

	Steps
	Check which version of the .NET Framework is installed on the machine hosting your SQL Server.
	Open Visual Studio and check that SSDT is installed.
	Create a new project.
	Add a new item to the project.
	Amend the template with your new code.
	Build the solution.
	Publish the solution.
	Open SSMS and check the new function appears.
	Create a Transact-SQL query using your new managed code function.

Lab: Implementing Managed Code in SQL Server

Scenario

You work for the rapidly expanding Adventure Works Bicycle Company Inc. A new developer has joined the database team and has decided to implement almost all of the logic in SQL CLR assemblies. You will determine if this is appropriate. You will also implement and test a supplied .NET assembly.

Objectives

After completing this lab you will be able to:

- Assess whether proposed functionality should be implemented in Transact-SQL, or CLR managed code.
- Implement a CLR assembly.
- Create a table-valued CLR function.

Estimated Time: 45 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Assessing Proposed CLR Code

Scenario

You first have to assess a list of proposed functionality for your company database. You will determine which functions should be implemented using Transact-SQL and which should be implemented using CLR managed code integrated into SQL Server.

The main tasks for this exercise are as follows:

1. Prepare the Environment
2. Review the List of Proposed Functionality

► Task 1: Prepare the Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab13\Starter** folder as Administrator.

► Task 2: Review the List of Proposed Functionality

1. Open **TSQL_or_Managed_Code.docx** from the **D:\Labfiles\Lab13\Starter** folder.
2. Review the proposed functionality and for each row, enter your recommended choice of implementation (SQL CLR or Transact-SQL) in column B and your reasons for that choice in column C.
3. Compare your choices with the answers in **TSQL_or_Managed_Code_Solution.docx**, in the **D:\Labfiles\Lab13\Solution** folder.
4. Close any open WordPad windows when you have finished.

Results: After completing this lab, you will have determined which type of code to use for each new feature.

Exercise 2: Creating a Scalar-Valued CLR Function

Scenario

You have been asked to create a scalar-valued function using CLR managed code. The function takes two parameters and performs a regular expression match on them. The first parameter is a text field to be searched and the other is a regular expression pattern. It returns 1 if a match is found, otherwise 0 is returned. Your function can be used in a WHERE clause of a SELECT statement.

The main tasks for this exercise are as follows:

1. Create a Scalar-Valued Function
2. Publish the Scalar-Valued Function
3. Create an Asymmetric Key and a Login in the Database
4. Sign and Publish the Assembly
5. Test the Scalar-Valued Function

► Task 1: Create a Scalar-Valued Function

1. Start SQL Server Management Studio, and then connect to the MIA-SQL Server instance by using Windows authentication.
2. Start Visual Studio.
3. Open **ClrPractice.sln** from the **D:\Labfiles\Lab13\Starter\ClrPractice** folder.
4. Open **IsRegexMatch.cs** and examine the code.
5. Build the solution, and check that the solution builds without errors.

► Task 2: Publish the Scalar-Valued Function

- Attempt to publish the assembly to the **MIA-SQL** Server instance and the **AdventureWorks2014** database.

Publishing fails with an error message. What is the problem?

► Task 3: Create an Asymmetric Key and a Login in the Database

- Open **Create_asymmetric_key.sql** from the **D:\Labfiles\Lab13\Starter** folder. Edit the script in the file to create an asymmetric key from **strong_name.snk** in the **D:\Labfiles\Lab13\Starter** folder, then to create a login called **sign_assemblies** from the asymmetric key. Grant the new login the UNSAFE ASSEMBLY permission. Execute the script when you have finished editing.

► Task 4: Sign and Publish the Assembly

- In Visual Studio, change the properties of the **ClrPractice** project to sign the assembly with **strong_name.snk** in the **D:\Labfiles\Lab13\Starter** folder. Rebuild the project, then publish the assembly to the **MIA-SQL** Server instance and the **AdventureWorks2014** database.

► Task 5: Test the Scalar-Valued Function

1. In SSMS, verify that the **ClrPractice** assembly is available in the **AdventureWorks2014** database.
2. Verify that the **IsRegexMatch** function is available in the **AdventureWorks2014** database.
3. Open **RegExMatch.sql** from the **D:\Labfiles\Lab13\Starter** folder.
4. Review and execute the query to verify that the function works as expected.
5. Close the file, but keep SSMS and Visual Studio open for the next exercise.

Results: After completing this exercise, you will have a scalar-valued CLR function available in SQL Server Management Studio.

Exercise 3: Creating a Table-Valued CLR Function

Scenario

You have also been asked to create a table-valued function using CLR managed code. The function takes two parameters and performs a regular expression match on them. The first parameter is a text field to be searched and the other is a regular expression pattern. This function returns a list of string values for the matches.

The main tasks for this exercise are as follows:

1. Create a Table-Valued function
2. Publish and Test the Table-Valued Function

► Task 1: Create a Table-Valued function

1. In Visual Studio, open **RegexMatchesTV.cs**, and examine the code.
2. Build the solution, and check that the solution builds without errors.

► Task 2: Publish and Test the Table-Valued Function

1. Publish the assembly to the MIA-SQL server instance and the **AdventureWorks2014** database.
2. In SSMS, verify that the **RegexMatches** function is available in the **AdventureWorks2014** database.
3. Open **TestRegExMatchex.sql** from the **D:\Labfiles\Lab13\Starter** folder.
4. Review and execute the queries to verify that the function works as expected.
5. If time permits, you could test the **StringAggregate** function by using **Test_StringAggregate.sql**.
6. Close SSMS without saving any changes.
7. Close Visual Studio without saving any changes.

Results: After completing this exercise, you will have a table-valued CLR function available in SQL Server Management Studio.

Question: After publishing managed code to a database, what do you think the issues are with using it?

Module Review and Takeaways

In this module, you have learned how to use CLR managed code to create user-defined database objects for SQL Server.

You should now be able to:

- Explain the importance of CLR integration in SQL Server.
- Implement and publish CLR assemblies using SQL Server Data Tools (SSDT).

Review Question(s)

Question: This module has reviewed the pros and cons of using managed code within a SQL Server database. You have integrated some prewritten C# functions into a database and tested them in some queries.

How might you use managed code in your own SQL Server environment? How do you assess the pros and cons for your specific situation?

Module 14

Storing and Querying XML Data in SQL Server

Contents:

Module Overview	14-1
Lesson 1: Introduction to XML and XML Schemas	14-2
Lesson 2: Storing XML Data and Schemas in SQL Server	14-11
Lesson 3: Implementing the XML Data Type	14-18
Lesson 4: Using the Transact-SQL FOR XML Statement	14-22
Lesson 5: Getting Started with XQuery	14-34
Lesson 6: Shredding XML	14-42
Lab: Storing and Querying XML Data in SQL Server	14-52
Module Review and Takeaways	14-57

Module Overview

XML provides rules for encoding documents in a machine-readable form. It has become a widely adopted standard for representing data structures, rather than sending unstructured documents. Servers that are running Microsoft® SQL Server® data management software often need to use XML to interchange data with other systems; many SQL Server tools provide an XML-based interface.

SQL Server offers extensive handling of XML, both for storage and querying. This module introduces XML, shows how to store XML data within SQL Server, and shows how to query the XML data.

The ability to query XML data directly avoids the need to extract data into a relational format before executing Structured Query Language (SQL) queries. To effectively process XML, you need to be able to query XML data in several ways: returning existing relational data as XML, and querying data that is already XML.

Objectives

After completing this module, you will be able to:

- Describe XML and XML schemas.
- Store XML data and associated XML schemas in SQL Server.
- Implement XML indexes within SQL Server.
- Use the Transact-SQL FOR XML statement.
- Work with basic XQuery queries.

Lesson 1

Introduction to XML and XML Schemas

Before you work with XML in SQL Server, this lesson provides an introduction to XML and how it is used outside SQL Server. You will learn some core XML-related terminology, along with how you can use schemas to validate and enforce the structure of XML.

This lesson also explores the appropriate uses for XML when you are working with SQL Server.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain core XML concepts.
- Explain the difference between fragments and documents.
- Describe the role of XML namespaces.
- Describe the role of XML schemas.
- Determine appropriate use cases for XML data storage in SQL Server.

Question: Do you currently work with applications that use XML? If your application does use XML, have you considered storing and processing that XML data on SQL Server?

Core XML Concepts

XML is a plain-text, Unicode-based metalanguage; that is, a language used to describe language. You can use it to hold both structured and semistructured data. It is not tied to any particular vendor, language, or operating system. It provides access to a wide range of technologies for manipulating, structuring, transforming, and querying data.

- XML is a plain-text, Unicode-based metalanguage
- It represents both structured and semistructured data
- It is not tied to any programming language, operating system, or vendor
- There are two approaches to encoding XML:
 - Element-centric
 - Attribute-centric

Data Interchange

XML came to prominence as a format for interchanging data between systems. It follows the same basic structure rules as other markup languages (such as HTML) and is used as a self-describing language.

Consider the following XML document:

XML Document

```
<?xml version="1.0" ?>
<?xml-stylesheet href="orders.xsl" type="text/xsl"?>
<orders>
    <order id="ord123456">
        <customer id="cust0921">
            <first-name>Dare</first-name>
            <last-name>Obasanjo</last-name>
            <address>
                <street>One Microsoft
                <city>Redmond</city>
                <state>WA</state>
            </address>
        </customer>
    </order>
</orders>
```

```

                <zip>98052</zip>
            </address>
        </customer>
    </order>
    <order id="ord123457">
        <customer id="cust0067">
            <first-name>Shai</first-name>
            <last-name>Bassli</last-name>
            <address>
                <street>567 3rd Ave</street>
                <city>Saginaw</city>
                <state>MI</state>
                <zip>53900</zip>
            </address>
        </customer>
    </order>
</orders>

```

Without any context and information, you can determine that this document holds the details about customer orders, the customers who placed the order, and the customer's name and address details. This explains why XML is defined as a self-describing language. In formal terminology, this is described as "deriving a schema" from a document.

XML Specifics

The lines in the example document that start with "<?" are referred to as processing instructions. These instructions are not part of the data, but determine the details of encoding. The first line in the preceding example is known as the prolog, and shows that version "1.0" of the XML specification is being used. The second line is a processing instruction that indicates the use of the extensible style sheet "orders.xsl" to format the document for display, if displaying the document becomes necessary.

The third line of the example is the first tag of the document and defines the "orders" element. Note that the document data starts with an opening orders element and finishes with a closing orders element shown as "</orders>." XML allows for repeating data, so the above example contains two orders for different customers.

 **Note:** XML elements are case-sensitive. For example, <street> is not the same as <Street>.

Element-Centric vs. Attribute-Centric XML

There are two ways to encode data in XML. The following example shows element-centric XML:

Element-Centric XML

```

<Supplier>
    <Name>Tailspin Toys</Name>
    <Rating>12</Rating>
</Supplier>

```

The following example shows the equivalent data in attribute-centric XML:

Attribute-Centric XML

```

<Supplier Name="Tailspin Toys" Rating="12">
</Supplier>

```

Note that, if all data for an element is contained in attributes, a shortcut form of element is available.

As an example, the following two XML elements are equivalent:

Attribute-Centric Shortcut

```
<Supplier Name="Tailspin Toys" Rating="12"></Supplier>
<Supplier Name="Tailspin Toys" Rating="12"/>
```

Using the above as an example, the most obvious benefit to the attribute-centric approach is the reduced size of the data. The element-centric approach needs 65 characters versus the 41 characters needed by the attribute-centric XML data—a large saving of 37 percent. However, element-centric XML is a better option in some circumstances, because it can describe more complex data; it can define an element as nullable; and the data can be parsed quicker, because only the elements need to be processed.

SQL Server can output XML encoded in either way; by using the FOR XML statement, you can produce XML that combines both these approaches.

Fragments vs. Documents

Well-formed XML has only one top-level, or root, element and element tags are correctly nested within each other. Text that has multiple top-level elements is considered a fragment, not a document.

Consider the following XML document:

XML Document

```
<order id="ord123456">
    <customer id="cust0921" />
</order>
```

- XML Document
 - One top-level root element
- XML Fragment
 - Multiple top-level elements
- Well-formed XML will include at least a prolog, one top-level element, and correctly nested element tags

This code provides the details for a single order and would be considered to be an XML document.

Now consider the following XML code:

XML Fragment

```
<order id="ord123456">
    <customer id="cust0921" />
</order>
<order id="ord123457">
    <customer id="cust0925" />
</order>
```

This text contains the details of multiple orders. Although it is perfectly reasonable XML, it is considered to be a fragment of XML rather than a document.

To be called a document, the XML needs to have a single root element, as shown in the following example:

Well-formed XML Document

```
<?xml version="1.0" ?>
<orders>
    <order id="ord123456">
        <customer id="cust0921" />
    </order>
    <order id="ord123457">
        <customer id="cust0925" />
    </order>
</orders>
```

Well-formed XML will also include at least a prolog defining which version of the XML specification is being used.

XML Namespaces

An XML namespace is a collection of names that you can use as element or attribute names. They are primarily used to avoid name conflicts on elements in XML documents.

Name Conflicts

This XML defines a table in HTML:

HTML Table

```
<table>
    <tr>
        <td>Chicago</td>
        <td>New York</td>
        <td>London</td>
        <td>Paris</td>
    </tr>
</table>
```

This XML defines a piece of furniture:

Table Furniture

```
<table>
    <name>Side Table</name>
    <length>80</length>
    <width>80</width>
    <height>100</height>
    <legs>4</legs>
</table>
```

- Primarily used to avoid element name conflicts
- One XML document can have many namespace definitions associated with it
- Namespaces are commonly attached to prefixes; if defined without a prefix, the namespace becomes the default for the XML document
- Defined as an XML attribute:
 - xmlns:URL

There would be a name conflict if an application required both these XML fragments to be contained within one XML document. XML has a mechanism to resolve these name conflicts with prefixes.

The previous two examples could be rewritten as:

XML Using Prefixes

```
<?xml version="1.0" ?>
<data>
    <html:table>
        <html:tr>
            <html:td>Chicago</html:td>
            <html:td>New York</html:td>
            <html:td>London</html:td>
            <html:td>Paris</html:td>
        </html:tr>
    </html:table>
    <furniture:table>
        <furniture:name>Side Table</furniture:name>
        <furniture:length>180</furniture:length>
        <furniture:width>80</furniture:width>
        <furniture:height>100</furniture:height>
        <furniture:legs>4</furniture:legs>
    </furniture:table>
</data>
```

The above XML has resolved the name conflict, but isn't valid XML until the prefixes are defined in a namespace.

XML Namespace

An XML namespace is defined by using the special attribute **xmlns**. The value of the attribute must be a valid Universal Resource Identifier (URI) or a Uniform Resource Name (URN). This namespace URI is most commonly a URL, which will point to a location on the Internet. This location does not need to link directly to an XML schema. You will see how XML schemas are related to namespaces in the next topic.

The following code provides examples of an XML namespace attribute:

XML Namespace Attributes

```
xmlns=http://schemas.microsoft.com/sqlserver/profiles/gml
xmlns:h=http://www.w3.org/TR/xhtml12/
```

The namespace attributes have to be added at the root element in the XML document, or they can be duplicated at each node element that requires them.



Best Practice: Industry best practice is to include namespace attributes in the top level node to reduce unnecessary duplication throughout the document.

To make the previous XML document well-formed, the prefixes need to have namespaces associated with them:

Using Namespaces

```
<?xml version="1.0" ?>
<data
  xmlns:html="http://www.w3.org/TR/html4/"
  xmlns:furniture="http://www.nopanet.org/?page=OFDAxmlCatalog">
    <html:table>
      <html:tr>
        <html:td>Chicago</html:td>
        <html:td>New York</html:td>
        <html:td>London</html:td>
        <html:td>Paris</html:td>
      </html:tr>
    </html:table>
    <furniture:table>
      <furniture:name>Side Table</furniture:name>
      <furniture:length>180</furniture:length>
      <furniture:width>80</furniture:width>
      <furniture:height>100</furniture:height>
      <furniture:legs>4</furniture:legs>
    </furniture:table>
  </data>
```

If a prefix isn't specified in the namespace attribute, that namespace will be used by default in any XML elements without a prefix.

XML Schemas

XML schemas are used to define the specific elements, attributes, and layout permitted within an XML document. A well-formed XML document is one that fulfills the criteria specified in the *Fragments vs. Documents* topic. If a well-formed XML document is validated against an XML schema, the document is said to be a valid and well-formed XML document.

The World Wide Web Consortium (W3C) defined XML schemas to be able to describe:

- Elements that can or must appear in a document.
- Attributes that can or must appear in a document.
- Which elements are child elements.
- The order of child elements.
- The number of child elements.
- Whether an element is empty or can include text.
- Data types for elements and attributes.
- Default and fixed values for elements and attributes.

- XML schema describes the structure of an XML document
- XML schema language is also called XML Schema Definition (XSD)
- An XML schema provides for the following:
 - Ability to validate constraints
 - Validation of the data by SQL Server
 - Data type information
 - Information about types of attributes and elements, the permitted structure of the XML

XML schemas are often referred to as XML Schema Definitions (XSDs). XSD is also the default file extension that most products use when they are storing XML schemas in files.

Example XML Schema

The first thing to consider is that an XSD is written in XML, and therefore can have a namespace and XSD to describe it. Put another way, an XSD can have an XSD that can be used to validate it.

This example XSD has a namespace that links to the W3C definition of XML Schemas:

Example XSD

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="manual">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="title" type="xsd:string"/>
                <xsd:element name="author" type="xsd:string"/>
                <xsd:element name="published" type="xsd:date"/>
                <xsd:element name="version" type="xsd:decimal"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

The above XML schema could be used to validate the following XML:

XML to be Validated

```
<manual>
    <title>How to use XML and SQL Server</title>
    <author>Stephen Jiang</author>
    <published>2016-05-07</published>
    <version>1.07</version>
</manual>
```

Appropriate Use of XML Data Storage in SQL Server

Given how widely XML has come to be used in application development in higher application tiers, there is a tendency to overuse XML within the database. It is important to consider when it is and is not appropriate to use XML within SQL Server.

XML vs. Objects

Higher-level programming languages that are used for constructing application programs often represent entities such as customers and orders as objects. Many developers see SQL Server as a simple repository for objects; that is, an object-persistence layer.

- Use Cases
 - Processing XML from external applications
 - Structure of the data is undefined
 - Improve the interoperability of the data
 - An explicit sequence within the data needs to be maintained
 - Indexing and improved querying is required on the XML

Consider the following table definition:

Table to Store XML Objects

```
CREATE TABLE ObjectStore (
    ObjectId uniqueidentifier PRIMARY KEY,
    PersistedData xml
);
```

There is no suggestion that this would make for a good database design, but note that you could use this table design to store all objects from an application—customers, orders, payments, and so on—in a single table. Compare this to how tables have been traditionally designed in relational databases.

SQL Server gives the developer a wide range of choices, from a simple XML design at one end of the spectrum, to fully normalized relational tables at the other end. Recognize that there is no generic answer for how a SQL Server database should be designed; instead, there's a range of options.

Example Use Cases

There are several reasons for storing XML data within SQL Server:

- You may be dealing with data that is already in XML, such as an order that you are receiving electronically from a customer. You may want to share, query, and modify the XML data in an efficient and transacted way.
- You may need to achieve a level of interoperability between your relational and XML data. Imagine that you have to join a customer table with a list of customer IDs that are being sent to you as XML.
- You might need to use XML formats to achieve cross-domain applications and to have maximum portability for your data. Other systems that you are communicating with may be based on entirely different technologies, and might not represent data in the same way as your database server.
- You might not know the structure of your data in advance. It is common to have a mixture of structured and semistructured data. A table might hold some standard relational columns, but also hold some less structured data in XML columns. For an example, see the HumanResources.JobCandidate table in the AdventureWorks database.
- You might need to preserve a sequence within your data. For example, you might need to retain order detail lines in a specific sequence. Relational tables and views have no implicit sequence. XML documents can exhibit a predictable sequence.
- You may want to have SQL Server validate that your XML data meets a particular XML schema before processing it.
- You might want to store transferred XML data for historical reasons, or archival purposes.
- You may want to create indexes on your XML data to support faster Business Information (BI) queries.

Demonstration: Introduction to XML and XML Schemas

In this demonstration, you will see how to structure XML and structure XML schemas.

Demonstration Steps

Structure XML and Structure XML Schemas

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **AdventureWorks\Student** with the password **Pa55w.rd**.
2. Navigate to **D:\Demofiles\Mod14**, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait until the script finishes.
4. On the taskbar, click **Microsoft SQL Server Management Studio**.
5. In the **Connect to Server** dialog box, in **Server name** box, type **MIA-SQL** and then click **Connect**.
6. On the **File** menu, point to **Open**, and then click **Project/Solution**.
7. In the **Open Project** dialog box, navigate to **D:\Demofiles\Mod14**, click **Demo14.ssmssln**, and then click **Open**.
8. In Solution Explorer, under **Miscellaneous**, and then double-click **XML_Sample_1.xml**.
9. Note the warning line under the second **<Production.Product>** tag. Position the cursor on the tag to display the warning.
10. Note that the XML editor in SSMS understands XML and formats it appropriately.
11. Note that the **Color** attribute is missing from elements where the data is NULL.
12. In Solution Explorer, under **Miscellaneous**, double-click **XML_Sample_2.xml**.
13. Note that this document contains a root element, so there is no longer a warning on the second **<Production.Product>** tag.
14. In Solution Explorer, under **Miscellaneous**, double-click **XML_Sample_3.xml**.
15. Note that this file contains an XML schema followed by the XML data.
16. Leave SSMS open for the next demonstration.

Lesson 2

Storing XML Data and Schemas in SQL Server

Now that you have learned about XML, schemas, and the surrounding terminology, you can consider how to store XML data and schemas in SQL Server. This is the first step in learning how to process XML effectively within SQL Server.

You need to see how the XML data type is used, how to define schema collections that contain XML schemas, how to declare both typed and untyped variables and database columns, and how to specify how well-formed and valid the XML data needs to be before it can be stored.

Lesson Objectives

After completing this lesson, you will be able to:

- Use the XML data type.
- Create XML schema collections.
- Declare variables and database columns as either untyped or typed XML.
- Choose whether XML fragments can be stored, rather than entire XML documents.

XML Data

SQL Server has a native data type for storing XML data. You can use it for variables, parameters, and columns in databases. SQL Server also exposes several methods that you can use for querying or modifying the stored XML data.

xml is a built-in data type for SQL Server. It is an intrinsic data type, which means that it is not implemented separately through managed code. The **xml** data type is limited to a maximum size of 2 GB. You can declare variables, parameters, and database columns by using the **xml** data type.

You can see a variable that has been declared by using the **xml** data type in the following code example:

XML Variable

```
DECLARE @Orders xml;
SET @Orders = '<Customer Name="Terry"><Order ID="231310" ProductID="12124"/></Customer>';
SELECT @Orders;
```

- SQL Server has a native data type for XML
- Enables you to store XML documents and fragments
- Can be used for columns, variables, or parameters
- Can be indexed
- Exposes methods to query and modify XML
- XML is stored efficiently in an internal format and returned in its canonical form

Canonical Form

Internally, SQL Server stores XML data in a format that makes it easy to process. It does not store the XML data in the same format as it was received in.

For example, consider the following code:

Canonical Form

```
DECLARE @Settings xml;
SET @Settings = '<Setup><Application Name="StartUpCleanup"
State="On"></Application><Application Name="Shredder" State="Off">Keeps
Spaces</Application></Setup>';
SELECT @Settings;
```

When the previous code is executed, the resulting XML is:

```
<Setup>
  <Application Name="StartUpCleanup" State="On" />
  <Application Name="Shredder" State="Off">Keeps      Spaces</Application>
</Setup>
```

Note that the output that is returned is logically equivalent to the input, but the output is not exactly the same as the input. For example, the first closing "</Application>" has been removed and replaced by a closing ">". Semantically, the two pieces of XML are identical, the returned XML is referred to as having been returned in a canonical or logically equivalent form.

If an exact copy of the XML has to be stored and retrieved from the database, consider storing the XML as a string in, for example, a nvarchar(max). However, using this approach means you will be unable to make use of the ability to create indexes on the XML and other methods.

For more information about xml, see Microsoft Docs:



<https://aka.ms/Yqkpcw>

XML Schema Collections

Although the **xml** data type will only store well-formed XML, you can further constrain the stored values by associating the data type with an XML schema collection.

In the first lesson, you learned how you can use XML schemas to constrain what you can store in an XML document. SQL Server does not store XML schemas as database objects. SQL Server has an XML SCHEMA COLLECTION object that holds a collection of XML schemas.

When you associate an XML SCHEMA COLLECTION object with an XML variable, parameter, or database column, the XML to be stored in that location needs to conform to at least one of the schemas that is contained in the schema collection.

- You can associate XSD schemas with an xml data type through an XML schema collection
- The XML schema collection:
 - Stores the imported XML schemas
 - Validates XML instances
 - Types the XML data as it is stored in the database
 - When an XSD is associated with XML it is said to be typed

XML Schemas

XML schemas are legible to humans at some level, but they are designed to be processed by computer systems. Even simple schemas tend to have quite a high level of complexity. Fortunately, you do not need to be able to read (or worse, write!) such schemas. Tools and utilities generally create XML schemas, and SQL Server can create them, too. You will see an example of this in a later lesson.

For example, consider the following XML schema:

XML Schema

```

<xsd:schema targetNamespace="urn:schemas-microsoft-com:sql:SqlRowSet1"
    xmlns:schema="urn:schemas-microsoft-com:sql:SqlRowSet1"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
    elementFormDefault="qualified">
    <xsd:import
        namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
        schemaLocation="http://schemas.microsoft.com/sqlserver/2004/sqltypes/sqltypes.xsd" />
    <xsd:element name="Production.Product">
        <xsd:complexType>
            <xsd:attribute name="ProductID"
                type="sqltypes:int" use="required" />
            <xsd:attribute name="Name" use="required">
                <xsd:simpleType>
                    <xsd:restriction
                        base="sqltypes:nvarchar"
                        sqltypes:localeId="1033" sqltypes:sqlCompareOptions=
                            "IgnoreCase IgnoreKanaType IgnoreWidth">
                        <xsd:maxLength value="50" />
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute name="Size">
                <xsd:simpleType>
                    <xsd:restriction
                        base="sqltypes:nvarchar"
                        sqltypes:localeId="1033" sqltypes:sqlCompareOptions=
                            "IgnoreCase IgnoreKanaType IgnoreWidth">
                        <xsd:maxLength value="5" />
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute name="Color">
                <xsd:simpleType>
                    <xsd:restriction
                        base="sqltypes:nvarchar"
                        sqltypes:localeId="1033" sqltypes:sqlCompareOptions=
                            "IgnoreCase IgnoreKanaType IgnoreWidth">
                        <xsd:maxLength value="15" />
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Creating an XML Schema Collection

An XML schema collection holds one or more schemas. The data that is being validated must match at least one of the schemas within the collection.

You create an XML schema collection by using the CREATE XML SCHEMA COLLECTION syntax that is shown in the following code snippet:

CREATE XML SCHEMA COLLECTION

```
CREATE XML SCHEMA COLLECTION SettingsSchemaCollection
AS
N'<?xml version="1.0" ?>
<xsd:schema
...
</xsd:schema>';
```

Altering Schema Collections

You can only modify a schema collection in a limited way using Transact-SQL. You can add new schema components to an existing schema collection by using the ALTER SCHEMA COLLECTION statement.

System Views

You can see the details of the existing XML schema collections by querying the **sys.xml_schema_collections** system view. You can see the details of the namespaces that are referenced by XML schema collections by querying the **sys.xml_schema_namespaces system** view. Like XML, XML schema collections are not stored in the format that you use to enter them. They are stripped into an internal format.

You can get an idea of how XML schema collections are stored by querying the **sys.xml_schema_components** system view, as shown in the following code example:

Storage of XML Schema Collections

```
SELECT sn.name namespace, scp.*
FROM sys.xml_schema_components scp
JOIN sys.xml_schema_collections sc ON scp.xml_collection_id = sc.xml_collection_id
JOIN sys.xml_schema_namespaces sn ON scp.xml_collection_id = sn.xml_collection_id;
```

Untyped vs. Typed XML

When you are storing XML data, you can choose to enable any XML to be stored, or you can choose to constrain the available values by associating the XML with an XML schema collection.

Untyped XML

You may choose to store any well-formed XML. One reason is that you might not have a schema for the XML data. Another reason is that you might want to avoid the processing overhead that is involved in validating the XML against the XML schema collection. For complex schemas, validating the XML can involve substantial work.

- Use the untyped XML data type in the following situations:
 - You do not have a schema for your XML data
 - You have schemas, but don't want the server to validate the data
- Use the typed XML data type in the following situations:
 - You have schemas and want the server to validate XML data
 - You want to take advantage of storage and query optimizations based on type information
 - You want to take advantage of type information during compilation of your queries

The following example shows the creation of a table that has an untyped XML column:

Untyped XML

```
CREATE TABLE App.Settings (
    SessionID int PRIMARY KEY,
    WindowSettings xml
);
```

You can store any well-formed XML in the WindowSettings column, up to the maximum size, which is currently 2 GB.

Typed XML

You may want to have SQL Server validate your data against a schema. You might want to take advantage of storage and query optimizations, based on the type information, or want to take advantage of this type information during the compilation of your queries.

The following example shows the same table being created, but this time, it has a typed XML column:

Typed XML

```
CREATE TABLE App.Settings (
    SessionID int PRIMARY KEY,
    WindowSettings xml (SettingsSchemaCollection)
);
```

In this case, a schema collection called SettingsSchemaCollection has been defined. SQL Server will not enable data to be stored in the WindowSettings column if it does not meet the requirements of at least one of the XML schemas in SettingsSchemaCollection.

CONTENT vs. DOCUMENT

While you are specifying typed XML, you can also specify whether it is necessary to provide entire XML documents or whether you can store XML fragments.

Using the CONTENT Keyword

This is equivalent to the previous definition because the CONTENT keyword is the default value for typed XML declarations.

- CONTENT is the default value
 - If nothing is specified, SQL Server will allow XML fragments to be stored
- DOCUMENT can be specified for typed XML
 - If an xml column has been linked to an XML Schema Collection, it can be defined as only allowing well-formed documents

CONTENT Keyword

```
CREATE TABLE App.Settings (
    SessionID int PRIMARY KEY,
    WindowSettings xml (CONTENT SettingsSchemaCollection)
);
```

SQL Server assumes XML data will be in the form of fragments, as opposed to documents, by default. This means that the preceding definition will have the same result as the Typed XML example Transact-SQL in the previous topic.

Using the DOCUMENT Keyword

The alternative to the default value of CONTENT is to specify the keyword DOCUMENT.

The DOCUMENT keyword is shown in the following code:

DOCUMENT Keyword

```
CREATE TABLE App.Settings (
    SessionID int PRIMARY KEY,
    WindowSettings xml (DOCUMENT SettingsSchemaCollection)
);
```

In this case, XML fragments could not be stored in the WindowSettings column. Only well-formed XML documents will be allowed. For example, a column that is intended to store a customer order can then be presumed to actually hold a customer order, and not some other type of XML document.

Demonstration: Working with Typed vs. Untyped XML

In this demonstration, you will see how to work with typed and untyped XML.

Demonstration Steps

Work with Typed and Untyped XML

1. Ensure that you have completed the previous demonstration.
2. In SSMS, in Solution Explorer, expand **Queries**, and then double-click **Demonstration 2.sql**.
3. Select the code under **Step 1**, and then click **Execute**.
4. Select the code under **Step 2**, and then click **Execute** to create an XML schema collection.
5. Select the code under **Step 3**, and then click **Execute** to create a table with a column that uses the collection.
6. Select the code under **Step 4**, and then click **Execute** to try to insert malformed XML. Note that the INSERT statement fails.
7. Select the code under **Step 5**, and then click **Execute** to try to insert well-formed XML that does not conform to the schema. Note that this INSERT statement fails too.
8. Select the code under **Step 6**, and then click **Execute** to insert a single row fragment.
9. Select the code under **Step 7**, and then click **Execute** to insert a multirow fragment.
10. Select the code under **Step 8**, and then click **Execute** to view the added XML data in the table.
11. Leave SSMS open for the next demonstration.

Check Your Knowledge

Question	
<p>Which of the following is not true about the xml data type?</p>	
<p>Select the correct answer.</p>	
	<p>It can store up to 2 GB of data.</p>
	<p>It can be stored with, or without, an associated XSD.</p>
	<p>It can only store well-formed XML documents.</p>
	<p>The CONTENT keyword is the default value for typed XML.</p>
	<p>It can be used for variables, parameters, and columns.</p>

Lesson 3

Implementing the XML Data Type

Indexes on XML columns are critical for achieving the high performance of XML-based queries. There are four types of XML index: a primary index and three types of secondary index. This lesson discusses how you can use each of them to achieve the maximum performance gain for your queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the need for XML indexes.
- Explain how to use each of the four types of XML index.

What Are XML Indexes?

Indexes are used in SQL Server to improve the performance of queries. XML indexes are used to improve the performance of XQuery-based queries. (XQuery will be discussed later in this module.)

Many systems query XML data directly as text. This can be very slow, particularly if the XML data is large. You saw earlier how XML data is not directly stored in a text format in SQL Server. For ease of querying, it is broken into a form of object tree that makes it easier to navigate in memory.

Rather than having to create these object trees as required for queries, which is also a relatively slow process, you can define XML indexes. An XML index is similar to a copy of an XML object tree that is saved into the database for rapid reuse.

It is important to note that XML indexes can be quite large, compared to the underlying XML data. Relational indexes are often much smaller than the tables on which they are built, but it is not uncommon to see XML indexes that are larger than the underlying data.

You should also consider alternatives to XML indexes. Promoting a value that is stored within the XML to a persisted calculated column would make it possible to use a standard relational index to quickly locate the value.

- XML data can be slow to access
 - XML indexes can help with query performance

Types of XML Indexes

SQL Server supports four types of XML index: a primary XML index and up to three secondary XML types.

Primary XML Index

The primary XML index provides a persisted object tree in an internal format. The tree has been formed from the structure of the XML, is used to speed up access to elements and attributes within the XML, and avoids the need to read the entire XML document for every query. Before you can create a primary XML index on a table, the table must have a clustered primary key. If a primary index isn't created, the database engine will need to create this object tree every time any query has to use the xml data.

Based on the App.Settings table that was used as an example earlier, you could create a primary XML index by executing the following code:

Primary XML Index

```
CREATE PRIMARY XML INDEX PXML_Settings_WindowSettings
ON App.Settings (WindowSettings);
```

Secondary XML Indexes

Most of the querying benefit comes from primary XML indexes, but SQL Server also enables the creation of three types of secondary XML index. These secondary indexes are designed to speed up a particular type of query. These are: PATH, VALUE, and PROPERTY:

- A PATH index helps to decide whether a particular path to an element or attribute is valid. It is typically used with the **exist()** XQuery method. (XQuery is discussed later in this module.)
- A VALUE index helps to obtain the value of an element or attribute.
- A PROPERTY index is used when retrieving multiple values through PATH expressions.

You can only create a secondary XML index after a primary XML index has been established.

When you are creating the secondary XML index, you need to reference the primary XML index.

Secondary XML Indexes

```
CREATE XML INDEX IXML_Settings_WindowSettings_Path
ON App.Settings (WindowSettings)
USING XML INDEX PXML_Settings_WindowSettings FOR PATH;

CREATE XML INDEX IXML_Settings_WindowSettings_Value
ON App.Settings (WindowSettings)
USING XML INDEX PXML_Settings_WindowSettings FOR VALUE;

CREATE XML INDEX IXML_Settings_WindowSettings_Property
ON App.Settings (WindowSettings)
USING XML INDEX PXML_Settings_WindowSettings FOR PROPERTY;
```

Primary XML index

- Provides a persisted object tree in an internal format that is used to speed access to elements and attributes within the XML
- Requires a clustered primary key on the table

Secondary XML index

- Can only be constructed after a primary XML index has been created
- You can construct three types of secondary indexes to help answer specific XQuery queries rapidly:
 - PATH
 - PROPERTY
 - VALUE

XML Tooling Support

Note that you can create both primary and secondary XML indexes in SQL Server Management Studio (SSMS), or by using Transact-SQL commands.

Demonstration: Implementing XML Indexes

In this demonstration, you will see how to implement XML indexes.

Demonstration Steps

1. Ensure that you have completed the previous demonstration.
2. In SSMS, in Solution Explorer, double-click **Demonstration 3.sql**.
3. Select the code under **Step 1**, and then click **Execute**.
4. Select the code under **Step 2**, and then click **Execute** to create a primary XML index.
5. Select the code under **Step 3**, and then click **Execute** to create a secondary VALUE index.
6. Select the code under **Step 4**, and then click **Execute** to query the **sys.xml_indexes** system view.
7. Select the code under **Step 5**, and then click **Execute** to drop and recreate the table without a primary key.
8. Select the code under **Step 6**, and then click **Execute** to try to add the primary **xml index** again. Note that this will fail.
9. Leave SSMS open for the next demonstration.

Categorize Activity

Categorize each statement against the appropriate index. Indicate your answer by writing the category number to the right of each statement.

Items	
1	Requires a clustered primary index to already exist on the table.
2	Requires a primary XML index to already exist.
3	Only one type.
4	Three different types.
5	Will provide the most performance improvement.
6	Could help improve XQueries on Values.
7	Avoids the need to parse the whole XML.
8	Could help improve XQueries on Properties.
9	Could help improve XQueries checking the existence of a node.

Category 1	Category 2
PRIMARY Index	SECONDARY Index

Lesson 4

Using the Transact-SQL FOR XML Statement

You have seen how SQL Server can store XML in its different formats. In this lesson, you will see how to retrieve XML from data stored in traditional tables and rows.

We often need to return data as XML documents, even though it is stored in relational database columns. Typically, this requirement relates to the exchange of data with other systems, including those from other organizations. When you add the FOR XML clause to a traditional Transact-SQL SELECT statement, it causes the output to be returned as XML instead of as a relational rowset.

SQL Server provides several modes for the FOR XML clause to enable the production of many styles of XML document. You will be learning about each of these modes and their related options.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the role of the FOR XML clause.
- Use RAW mode queries.
- Use AUTO mode queries.
- Use EXPLICIT mode queries.
- Use PATH mode queries.
- Retrieve nested XML.

Introduction to the FOR XML Clause

You can use the FOR XML clause to enable the Transact-SQL SELECT statement to return data in an XML format. You can configure it to return the attributes, elements, and/or schema that are required by client applications.

The FOR XML clause can be used in one of four modes:

1. **RAW** mode generates a single element per row in the rowset that the SELECT statement returns.
2. **AUTO** mode generates nesting in the resulting XML, based on the way in which the SELECT statement is specified. You have minimal control over the shape of the XML that is generated. If you need to produce nested XML, AUTO mode is a better choice than RAW mode.
3. **EXPLICIT** mode gives you more control over the shape of the XML. You can use it when other modes do not provide enough flexibility, but this is at the cost of greater complexity. In deciding the shape of the XML, you can mix attributes and elements as you like.
4. **PATH** mode, together with the nested FOR XML query capability, provides much of the flexibility of the EXPLICIT mode in a simpler manner.

- Extends Transact-SQL SELECT syntax
- Returns data as XML instead of rows and columns
- Is configurable to return attributes, elements, and the schema
- Has four different modes:
 - RAW
 - AUTO
 - EXPLICIT
 - PATH

Each of these modes are covered in more detail in the next topics.

For more information about FOR XML, see Microsoft Docs:

FOR XML (SQL Server)

<https://aka.ms/A74x8t>

Using RAW Mode Queries

RAW mode is the simplest mode to work with in the FOR XML clause. It returns a simple XML representation of the rowset and can optionally specify a row element name and a root element.

Consider this simple Transact-SQL query:

Traditional SELECT Statement

```
SELECT TOP 5 FirstName, LastName,
PersonType
FROM Person.Person
ORDER BY FirstName, LastName;
```

FirstName	LastName	PersonType
A.	Leonetti	SC
A.	Wright	GC
A. Scott	Wright	EM
Aaron	Adams	IN
Aaron	Alexander	IN

- Enables SQL Server to return data as XML
- Can produce element- or attribute-centric XML
- Default is to produce XML fragments
- Does not include NULL columns
- Has additional options to:
 - Add a root node
 - Rename elements
 - Generate an XML schema
 - Include NULL data with the XSINIL option

Now consider the modified statement after adding the FOR XML clause:

Using RAW Mode in the FOR XML Clause

```
SELECT TOP 5 FirstName, LastName, PersonType
FROM Person.Person
ORDER BY FirstName, LastName
FOR XML RAW;
```

```
<row FirstName="A." LastName="Leonetti" PersonType="SC" />
<row FirstName="A." LastName="Wright" PersonType="GC" />
<row FirstName="A. Scott" LastName="Wright" PersonType="EM" />
<row FirstName="Aaron" LastName="Adams" PersonType="IN" />
<row FirstName="Aaron" LastName="Alexander" PersonType="IN" />
```

Note that one XML `<row>` element is returned for each row from the rowset, the element has a generic name of row, and all columns are returned as attributes. The returned order is based on the ORDER BY clause.

You can override the default row name in the XML and to add a root element:

Making a Well-formed XML Document

```
SELECT TOP 5 FirstName, LastName, PersonType
FROM Person.Person
ORDER BY FirstName, LastName
FOR XML RAW('Person'),
        ROOT('People');
```

This generates the following results:

```
<People>
<Person FirstName="A." LastName="Leonetti" PersonType="SC" />
<Person FirstName="A." LastName="Wright" PersonType="GC" />
<Person FirstName="A. Scott" LastName="Wright" PersonType="EM" />
<Person FirstName="Aaron" LastName="Adams" PersonType="IN" />
<Person FirstName="Aaron" LastName="Alexander" PersonType="IN" />
</People>
```

Element-Centric XML

You will notice that, in the previous examples, the columns from the rowset have been returned as attribute-centric XML. You can modify this behavior to produce element-centric XML by adding the ELEMENTS keyword to the FOR XML clause.

You can see this in the following query:

Element-Centric XML

```
SELECT TOP 5 FirstName, LastName, PersonType
FROM Person.Person
ORDER BY FirstName, LastName
FOR XML RAW('Person'),
        ROOT('People'),
        ELEMENTS;
```

When this query is executed, it returns the following output:

```
<People>
<Person>
  <FirstName>A.</FirstName>
  <LastName>Leonetti</LastName>
  <PersonType>SC</PersonType>
</Person>
<Person>
  <FirstName>A.</FirstName>
  <LastName>Wright</LastName>
  <PersonType>GC</PersonType>
</Person>
<Person>
  <FirstName>A. Scott</FirstName>
  <LastName>Wright</LastName>
  <PersonType>EM</PersonType>
</Person>
<Person>
  <FirstName>Aaron</FirstName>
  <LastName>Adams</LastName>
  <PersonType>IN</PersonType>
</Person>
<Person>
  <FirstName>Aaron</FirstName>
  <LastName>Alexander</LastName>
  <PersonType>IN</PersonType>
</Person>
```

```
</People>
```

Note that all the columns have now been returned as elements.

Adding an XML Schema

Another option that can be added to a RAW mode statement is the inclusion of an XML Schema:

Include XML Schema

```
SELECT TOP 5 FirstName, LastName, PersonType
FROM Person.Person
ORDER BY FirstName, LastName
FOR XML RAW('Person'),
        ROOT('People'),
        ELEMENTS,
        XMLSCHEMA('urn:schema_example.com');
```

The above Transact-SQL will result in the following XML being generated:

```
<People>
  <xsd:schema targetNamespace="urn:schema_example.com"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
  elementFormDefault="qualified">
    <xsd:import namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
    schemaLocation="http://schemas.microsoft.com/sqlserver/2004/sqltypes/sqltypes.xsd" />
    <xsd:element name="Person">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="FirstName">
            <xsd:simpleType sqltypes:sqlTypeAlias="[AdventureWorks].[dbo].[Name]">
              <xsd:restriction base="sqltypes:nvarchar" sqltypes:localeId="1033"
                sqltypes:sqlCompareOptions="IgnoreCase IgnoreKanaType IgnoreWidth"
                sqltypes:sqlSortId="52">
                <xsd:maxLength value="50" />
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="LastName">
              <xsd:simpleType sqltypes:sqlTypeAlias="[AdventureWorks].[dbo].[Name]">
                <xsd:restriction base="sqltypes:nvarchar" sqltypes:localeId="1033"
                  sqltypes:sqlCompareOptions="IgnoreCase IgnoreKanaType IgnoreWidth"
                  sqltypes:sqlSortId="52">
                <xsd:maxLength value="50" />
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="PersonType">
              <xsd:simpleType>
                <xsd:restriction base="sqltypes:nchar" sqltypes:localeId="1033"
                  sqltypes:sqlCompareOptions="IgnoreCase IgnoreKanaType IgnoreWidth"
                  sqltypes:sqlSortId="52">
                <xsd:maxLength value="2" />
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
<Person xmlns="urn:schema_example.com">
  <FirstName>A.</FirstName>
  <LastName>Leonetti</LastName>
  <PersonType>SC</PersonType>
</Person>
<Person xmlns="urn:schema_example.com">
```

```

<FirstName>A.</FirstName>
<LastName>Wright</LastName>
<PersonType>GC</PersonType>
</Person>
<Person xmlns="urn:schema_example.com">
    <FirstName>A. Scott</FirstName>
    <LastName>Wright</LastName>
    <PersonType>EM</PersonType>
</Person>
<Person xmlns="urn:schema_example.com">
    <FirstName>Aaron</FirstName>
    <LastName>Adams</LastName>
    <PersonType>IN</PersonType>
</Person>
<Person xmlns="urn:schema_example.com">
    <FirstName>Aaron</FirstName>
    <LastName>Alexander</LastName>
    <PersonType>IN</PersonType>
</Person>
</People>

```

Using Auto Mode Queries

By default, when using AUTO mode, each row in the result set is represented as an XML element that is named after the table (or alias) from which it was selected. AUTO mode generates nesting in the resulting XML, based on the way in which the SELECT statement is specified. You have minimal control over the shape of the XML that is generated. AUTO mode queries are more capable of dealing with nested XML, compared to the RAW mode.

- Enables SQL Server to return data as nested XML elements
- Can produce element- or attribute-centric XML
- Default is to produce XML fragments
- Does not include NULL columns
- Has the same additional options as RAW mode:
 - Add a root node
 - Rename elements by using aliases
 - Generate an XML schema
 - Include NULL data with the XSINIL option

FOR XML AUTO

AUTO mode queries are useful if you want to generate simple hierarchies, but they provide limited control of the resultant XML. If you need more control over the resultant XML than AUTO mode queries provide, you need to consider using the PATH or EXPLICIT modes instead. These are both covered in the next topics.

Let's look at the previous SQL example using the AUTO mode:

Using AUTO Mode in the FOR XML Clause

```

SELECT TOP 5 FirstName, LastName, PersonType
FROM Person.Person
ORDER BY FirstName, LastName
FOR XML AUTO;

```

Each table in the FROM clause, from which at least one column is listed in the SELECT clause, is represented as an XML element. The columns that are listed in the SELECT clause are mapped to attributes. You can see the output of the previous query below:

```

<Person.Person FirstName="A." LastName="Leonetti" PersonType="SC" />
<Person.Person FirstName="A." LastName="Wright" PersonType="GC" />
<Person.Person FirstName="A. Scott" LastName="Wright" PersonType="EM" />
<Person.Person FirstName="Aaron" LastName="Adams" PersonType="IN" />
<Person.Person FirstName="Aaron" LastName="Alexander" PersonType="IN" />

```

Note how the name of the table is directly used as the element name.

For this reason, it is common to provide an alias for the table, as shown in the following code:

Using a Table Alias

```
SELECT TOP 5 FirstName, LastName, PersonType
FROM Person.Person AS Person
ORDER BY FirstName, LastName
FOR XML AUTO;
```

This results in the following:

```
<Person FirstName="A." LastName="Leonetti" PersonType="SC" />
<Person FirstName="A." LastName="Wright" PersonType="GC" />
<Person FirstName="A. Scott" LastName="Wright" PersonType="EM" />
<Person FirstName="Aaron" LastName="Adams" PersonType="IN" />
<Person FirstName="Aaron" LastName="Alexander" PersonType="IN" />
```

To generate the well-formed XML that was produced in the previous topic, you can add the ROOT('People') option to the FOR XML AUTO statement.

Creating Nested XML

The benefits of using AUTO over the RAW mode is that SQL Server will return XML data nested in accordance with heuristics in the SELECT statement.

Generating More Complex XML

```
SELECT TOP 3 FirstName, LastName, City
FROM Person.Person AS Employee
INNER JOIN Person.BusinessEntityAddress AS BA
ON BA.BusinessEntityID = Employee.BusinessEntityID
INNER JOIN Person.Address AS Address
ON BA.AddressID = Address.AddressID
ORDER BY FirstName, LastName
FOR XML AUTO, ROOT ('Employees'), ELEMENTS;
```

Executing the above will result in the following XML being produced:

```
<Employees>
<Employee>
  <FirstName>A. Scott</FirstName>
  <LastName>Wright</LastName>
  <Address>
    <City>Newport Hills</City>
  </Address>
</Employee>
<Employee>
  <FirstName>Aaron</FirstName>
  <LastName>Adams</LastName>
  <Address>
    <City>Downey</City>
  </Address>
</Employee>
<Employee>
  <FirstName>Aaron</FirstName>
  <LastName>Alexander</LastName>
  <Address>
    <City>Kirkland</City>
  </Address>
</Employee>
</Employees>
```

Notice how the City value is nested inside the Address element, which in turn is nested inside the Employee element.

Using Explicit Mode Queries

EXPLICIT mode gives you the greatest control over the resulting XML, but at the price of query complexity. Many common queries that required EXPLICIT mode in earlier versions of SQL Server can be implemented by using PATH mode. (PATH mode was introduced in SQL Server 2005 and will be explained in the next topic.)

- Explicit mode
 - Enables tabular representation of XML documents
 - Enables complete control of the XML format
 - Requires complex SQL statements that can be more easily generated with other modes

FOR XML EXPLICIT

EXPLICIT mode queries define XML fragments as a universal table, which consists of a column for each piece of data that you require, and two additional columns. The additional columns are used to define the metadata for the XML fragment. The Tag column uniquely identifies the XML tag that will be used to represent each row in the results, and the Parent column is used to control the nesting of elements. Each row of data in the universal table represents an element in the resulting XML document.

The power of EXPLICIT mode is to mix attributes and elements at will, create wrappers and nested complex properties, create space-separated values (for example, the OrderID attribute may have a list of order ID values), and create mixed contents.

PATH mode, together with the nesting of FOR XML queries and the TYPE clause, gives enough power to replace most of the EXPLICIT mode queries in a simpler, more maintainable way. EXPLICIT mode is rarely needed now.

To produce XML in a similar format to the previous topic, the following SQL is required:

EXPLICIT Mode Example

```

SELECT 1 AS Tag,
       NULL AS Parent,
       Employee.FirstName AS [Employee!1!FirstName!ELEMENT],
       Employee.LastName AS [Employee!1!LastName!ELEMENT],
       NULL AS [Address!2!City!ELEMENT]
  FROM Person.Person AS Employee
 INNER JOIN Person.BusinessEntityAddress AS BA
    ON BA.BusinessEntityID = Employee.BusinessEntityID
 INNER JOIN Person.Address AS Address
    ON BA.AddressID = Address.AddressID
 UNION ALL
SELECT 2 AS Tag,
       1 AS Parent,
       Employee.FirstName,
       Employee.LastName,
       Address.City
  FROM Person.Person AS Employee
 INNER JOIN Person.BusinessEntityAddress AS BA
    ON BA.BusinessEntityID = Employee.BusinessEntityID
 INNER JOIN Person.Address AS Address
    ON BA.AddressID = Address.AddressID
 ORDER BY [Employee!1!FirstName!ELEMENT], [Employee!1!LastName!ELEMENT]
FOR XML EXPLICIT, ROOT('Employees');
```

Using Path Mode Queries

PATH mode provides a simpler way to mix elements and attributes. You can use it in many situations as an easier way to write queries than by using EXPLICIT mode.

FOR XML PATH

PATH mode is a simpler way to introduce additional nesting for representing complex properties. In PATH mode, column names or column aliases are treated as XML Path Language (XPath) expressions. (More detail on XPath will be provided later in this module.) These expressions indicate how the values are being mapped to XML. Each XPath expression is a relative XPath that provides the item type, such as the attribute, element, and scalar value, and the name and hierarchy of the node that will be generated relative to the row element.

You can use FOR XML EXPLICIT mode queries to construct such XML from a rowset, but PATH mode provides a simpler alternative to the potentially time-consuming EXPLICIT mode queries.

You can use PATH mode, together with the ability to write nested FOR XML queries and the TYPE directive to return xml data type instances, to write less complex queries. This gives enough power to replace most of the EXPLICIT mode queries in a simpler, more maintainable way.

Here is the SQL to produce the XML in the previous topic:

PATH Mode Example

```
SELECT TOP 3 FirstName, LastName, City AS "Address/City"
FROM Person.Person AS Employee
INNER JOIN Person.BusinessEntityAddress AS BA
ON BA.BusinessEntityID = Employee.BusinessEntityID
INNER JOIN Person.Address AS Address
ON BA.AddressID = Address.AddressID
ORDER BY FirstName, LastName
FOR XML PATH ('Employee'), ROOT ('Employees'), ELEMENTS;
```

- Uses XML Path Language (X Path) to specify XML format
- Enables the creation of nested data and specifies what should be exposed as an element or an attribute
- Column aliases determine the output:
 - @, the column will be added as an attribute
 - / will describe the hierarchy of the element
 - The two can be combined
- Easier to use than EXPLICIT mode

The XPath expressions can be used to control the structure of the XML. You can modify the default PATH behavior by using the “at” (@) symbol to define attributes or the forward slash (/) to define the hierarchy.

Let's add the address ID as an attribute in the previous example:

Using the @ Symbol

```
SELECT TOP 3 FirstName, LastName, Address.AddressID AS 'Address/@AddressID', City AS
"Address/City"
FROM Person.Person AS Employee
INNER JOIN Person.BusinessEntityAddress AS BA
ON BA.BusinessEntityID = Employee.BusinessEntityID
INNER JOIN Person.Address AS Address
ON BA.AddressID = Address.AddressID
ORDER BY FirstName, LastName
FOR XML PATH ('Employee'), ROOT ('Employees'), ELEMENTS;
```

This results in the following XML being produced:

```
<Employees>
  <Employee>
    <FirstName>A. Scott</FirstName>
    <LastName>Wright</LastName>
    <Address AddressID="250">
      <City>Newport Hills</City>
    </Address>
  </Employee>
  <Employee>
    <FirstName>Aaron</FirstName>
    <LastName>Adams</LastName>
    <Address AddressID="25953">
      <City>Downey</City>
    </Address>
  </Employee>
  <Employee>
    <FirstName>Aaron</FirstName>
    <LastName>Alexander</LastName>
    <Address AddressID="14543">
      <City>Kirkland</City>
    </Address>
  </Employee>
</Employees>
```

Note the use of the "/" to define the structure of the returned XML, and that both the "@" and "/" can be combined in aliases. If the alias for the Address.AddressID did not include the "/", the AddressID attribute would have been added to the Employee element. For example:

```
<Employee AddressID="250">
  <FirstName>A. Scott</FirstName>
  <LastName>Wright</LastName>
  <Address>
    <City>Newport Hills</City>
  </Address>
</Employee>
```

Nested FOR XML Query

Combining all the previous topics into a nested query:

Combining AUTO and PATH Queries

```
SELECT TOP 3 BusinessEntityID AS '@ID', LoginID AS '@Login',
       (SELECT FirstName, LastName
        FROM Person.Person AS EmployeeName
        WHERE EmployeeName.BusinessEntityID = Employee.BusinessEntityID
              ORDER BY FirstName, LastName
        FOR XML AUTO, TYPE, ELEMENTS)
  FROM HumanResources.Employee
  FOR XML PATH ('Employee'), ROOT ('Employees'), ELEMENTS;
```

You can use any combination of RAW, AUTO and PATH queries. The previous SQL example will produce this output:

```
<Employees>
  <Employee ID="225" Login="adventure-works\alan0">
    <EmployeeName>
      <FirstName>Alan</FirstName>
      <LastName>Brewer</LastName>
    </EmployeeName>
  </Employee>
  <Employee ID="193" Login="adventure-works\alejandro0">
    <EmployeeName>
```

```

<FirstName>Alejandro</FirstName>
<LastName>McGuel</LastName>
</EmployeeName>
</Employee>
<Employee ID="163" Login="adventure-works\alex0">
<EmployeeName>
<FirstName>Alex</FirstName>
<LastName>Nayberg</LastName>
</EmployeeName>
</Employee>
</Employees>

```

Retrieving Nested XML

You can use the **TYPE** keyword to return FOR XML subqueries as **xml** data types, rather than as **nvarchar** data types.

TYPE Keyword

In the previous topics in this lesson, you have seen how FOR XML AUTO queries can return attribute-centric or element-centric XML. If this data is returned from a subquery, it needs to be returned as a specific data type.

- Nested FOR XML statements with a TYPE option will create data in xml format
- If TYPE isn't used, the data is returned as characters
- You can nest FOR XML statements inside other FOR XML statements to create complex XML structures
- TYPE can be used on all modes of the FOR XML statement

The FOR XML clause was introduced in SQL Server 2000. That version of SQL Server did not have an xml data type. For that reason, subqueries that had FOR XML clauses had no way to return the xml data type. FOR XML subqueries in SQL Server 2000 returned the nvarchar data type instead.

SQL Server 2005 introduced the xml data type, but for backward compatibility, the data type for return values from FOR XML subqueries was not changed to xml. However, a new keyword, **TYPE**, was introduced that changes the return data type of FOR XML subqueries to xml.

You can use a nested FOR XML query, in a FOR XML query, to build nested XML.

Combining AUTO with TYPE Nested in a PATH Query

```

SELECT TOP 3 BusinessEntityID AS '@ID', LoginID AS '@Login',
(SELECT FirstName, LastName
FROM Person.Person AS EmployeeName
WHERE EmployeeName.BusinessEntityID = Employee.BusinessEntityID
ORDER BY FirstName, LastName
FOR XML AUTO, TYPE, ELEMENTS)
FROM HumanResources.Employee
FOR XML PATH ('Employee'), ROOT ('Employees'), ELEMENTS;

```

You can use any combination of RAW, AUTO and PATH queries. The previous SQL example will produce this output:

```

<Employees>
<Employee ID="225" Login="adventure-works\alan0">
<EmployeeName>
<FirstName>Alan</FirstName>
<LastName>Brewer</LastName>
</EmployeeName>
</Employee>
<Employee ID="193" Login="adventure-works\alejandro0">
<EmployeeName>

```

```

<FirstName>Alejandro</FirstName>
<LastName>McGue1</LastName>
</EmployeeName>
</Employee>
<Employee ID="163" Login="adventure-works\alex0">
    <EmployeeName>
        <FirstName>Alex</FirstName>
        <LastName>Nayberg</LastName>
    </EmployeeName>
</Employee>
</Employees>

```

Another type of nested XML query might be where XML is required in the results as an actual column containing XML alongside non-XML columns.

XML and Non-XML Results

```

SELECT Customer.CustomerID, Customer.TerritoryID,
    (SELECT SalesOrderID, [Status]
        FROM Sales.SalesOrderHeader AS soh
        WHERE Customer.CustomerID = soh.CustomerID
        FOR XML AUTO, TYPE) as Orders
    FROM Sales.Customer as Customer
    WHERE EXISTS
        (SELECT 1 FROM Sales.SalesOrderHeader AS soh
            WHERE soh.CustomerID = Customer.CustomerID)
    ORDER BY Customer.CustomerID;

```

Executing the previous Transact-SQL will return a table of results, with the last column containing the required XML data. This will be hyperlinked; if it was a varchar, the results would appear as plain text.

CustomerID	TerritoryID	Orders
11000	9	<soh SalesOrderID="43793" Status="5" /><soh SalesOrderID="51522" Status="5" /><soh SalesOrderID="57418" Status="5" />
11001	9	<soh SalesOrderID="43767" Status="5" /><soh SalesOrderID="51493" Status="5" /><soh SalesOrderID="72773" Status="5" />
11002	9	<soh SalesOrderID="43736" Status="5" /><soh SalesOrderID="51238" Status="5" /><soh SalesOrderID="53237" Status="5" />

Demonstration: FOR XML Queries

In this demonstration, you will see how to use FOR XML queries.

Demonstration Steps

1. Ensure that you have completed the previous demonstration.
2. In SSMS, in Solution Explorer, double-click **Demonstration 4.sql**.
3. Select the code under **Step 1**, and then click **Execute**.
4. Select the code under **Step 2**, click **Execute** to execute RAW mode queries, and then review the results.
5. Select the code under **Step 3**, click **Execute** to execute an AUTO mode query, and then review the results.
6. Select the code under **Step 4**, click **Execute** to execute an EXPLICIT mode query, and then review the results.
7. Select the code under **Step 5**, click **Execute** to execute PATH mode queries, and then review the results.
8. Select the code under **Step 6**, click **Execute** to execute a query using TYPE, and then review the results.
9. Select the code under **Step 7**, click **Execute** to run the same query without using the TYPE keyword, and then compare the results with those obtained in the previous step.
10. Leave SSMS open for the next demonstration.

Check Your Knowledge

Question
Which of the following is not true about RAW mode queries?
Select the correct answer.
<input type="checkbox"/> By default, they produce XML fragments.
<input type="checkbox"/> They produce nested XML based on SELECT heuristics.
<input type="checkbox"/> A root node can be added with the ROOT option.
<input type="checkbox"/> XSINIL will result in NULL columns being added to the results.

Lesson 5

Getting Started with XQuery

XQuery allows you to query XML data. Sometimes data is already in XML and you need to query it directly. You might want to extract part of the XML into another XML document; you might want to retrieve the value of an element or attribute; you might want to check whether an element or attribute exists; and finally, you might want to directly modify the XML. XQuery methods make it possible to perform these tasks.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the role of XQuery.
- Use the **query()** method.
- Use the **value()** method.
- Use the **exist()** method.
- Use the **modify()** method.

What is XQuery?

XQuery is a query language that is designed to query XML documents. It also includes elements of other programming languages, such as looping constructs.

XQuery was developed by a working group within the World Wide Web Consortium. It was developed in conjunction with other work in the W3C, in particular, the definition of Extensible Stylesheet Language Transformations (XSLT). XSLT makes use of a subset of XQuery that is known as XPath.

XPath is the syntax that is used to provide an address for specific attributes and elements within an XML document. You saw basic examples of this when you were considering FOR XML PATH mode queries in the previous lesson.

Look at the following XPath expression:

XPath Expression

```
/SalesHistory/Sale[@InvoiceNo=635]
```

- A querying language for XML
- Includes other language elements, for example looping
- Supports FLWOR
 - For, Let, Where, Order, and Return
- Has five methods
 - query(), value(), exist(), modify(), and nodes()

This XPath expression specifies a need to traverse the **SalesHistory** node—that is the root element because the expression starts with a slash mark (/)—then traverse the **Sale** subelements (note that there may be more than one of these), and then to access the **InvoiceNo** attribute. All invoices that have an invoice number attribute equal to 635 are returned.

Although there is unlikely to be more than one invoice with the number 635, nothing about XML syntax (without a schema) enforces this. One thing that can be hard to get used to with the XPath syntax is that you constantly need to specify that you want the first entry of a particular type—even though logically

you may think that it should be obvious that there would only be one. You indicate the first entry in a list by the expression [1].

To return the first sales record if there are more than one with an invoice number equal to 635:

Find the First Element in an XML Document

```
/SalesHistory/Sale[@InvoiceNo=635][1]
```

In XPath, you indicate attributes by using the “at” (@) prefix. The content of the element itself is referred to by the token **text()**.

FLWOR Expressions

In addition to basic path traversal, XPath supports an iterative expression language that is known as FLWOR and commonly pronounced “flower.” FLWOR stands for “for, let, where, order, and return,” which are the basic operations in a FLWOR query.

An example of a FLWOR expression is shown in the following XQuery **query()** method:

FLWOR Expression

```
SELECT @xmlDoc.query
(
<OrderedItems>
{
    for $i in /InvoiceList/Invoice/Items/Item
    return $i
}
</OrderedItems>
');
```

This query supplies **OrderedItems** as an element. Then, within that element, it locates all items on all invoices that are contained in the XML document and displays them as subelements of the **OrderedItems** element. An example of what the output may look like from this query is shown here:

```
<OrderedItems>
    <Item Product="1" Price="1.99" Quantity="2" />
    <Item Product="3" Price="2.49" Quantity="1" />
    <Item Product="1" Price="1.99" Quantity="2" />
</OrderedItems>
```

Note that becoming proficient at XQuery is an advanced topic that is beyond the scope of this course. The aim of this lesson is to make you aware of what is possible when you are using XQuery methods. The available XQuery methods are shown in the following table:

Method	Description
query()	This method returns untyped XML; the XML is selected by an XQuery expression.
value()	This method returns a scalar value; it takes XQuery and a SQL Type as its parameters.
exist()	This method returns a bit value; 1 if a node is found to exist; 0 if a node isn't found for the specific XQuery expression.
modify()	This method modifies the contents of an XML document based on the XML DML expression.
nodes()	This method can be used to shred XML into relational data.

Shredding XML is covered in more detail in the next lesson, as is the **nodes()** method.

Advantages of XQuery:

- It is easy to learn if you know SQL and XPath.
- When queries are written in XQuery, they require less code, compared to queries written in XSLT.
- XQuery can be used as a strongly typed language when the XML data is typed; this can improve the performance of the query by avoiding implicit type casts and provide type assurances that can be used when performing query optimization.
- XQuery can be used as a weakly typed language for untyped data to provide high usability. SQL Server implements static type inferencing with support for both strong and weak type relationships.
- XQuery 3.0 became a W3C recommendation on April 8, 2014, and will be supported by major database vendors. SQL Server currently supports the W3C version XQuery 1.0.

For more information about XQuery, see Microsoft Docs:

 **XQuery Language Reference (SQL Server)**

<https://aka.ms/Bhjwdn>

The **query()** Method

You can use the **query()** method to extract XML from an existing XML document. This document could be stored in an XML variable or database column.

The XML that is generated can be a subset of the original XML document. Alternatively, you could generate entirely new XML based on the values that are contained in the original XML document.

You can use the **query()** method to return untyped XML. The **query()** method takes an XQuery expression that evaluates to a list of XML nodes and enables users to create output XML, based in some way on the fragments that it extracts from the input XML.

An XQuery expression in SQL Server consists of two sections: a prolog and a body. The prolog can contain a namespace declaration. You will see how to do this later in this module. The body of an XQuery expression contains query expressions that define the result of the query. Both the input and output of a **query()** method are XML.

Note that, if NULL is passed to a **query()** method, the result that the method returns is also NULL.

- Returns untyped XML
- Expects a valid XQuery as input
- Can include the FLWOR statements
- Example:
 - `SELECT Demographics.query('StoreSurvey/NumberEmployees')`
FROM Sales.Store;

Example of a **query()** Method

A query to return the revenue and number of staff from the AdventureWorks example database:

query() Method

```
WITH XMLNAMESPACES ( DEFAULT 'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/StoreSurvey' )
SELECT Top 3 BusinessEntityID,
       Demographics.query('(/StoreSurvey/AnnualRevenue)') AS Revenue,
       Demographics.query('(/StoreSurvey/NumberEmployees)') As Staff
FROM   Sales.Store;
```

This query tells SQL Server to return the business entity id, the annual revenue, and the number of staff for every row in the Sales.Store table. Do not be too concerned with the namespace declaration in this example; because the XML document in this column has a defined namespace, the **query()** method needs to be made aware of it. Running the previous example will return rows in the following format:

BusinessEntityID	Revenue	Staff
292	<p1:AnnualRevenue xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey">80000</p1:AnnualRevenue>	<p1:NumberEmployees xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey">13</p1:NumberEmployees>
294	<p1:AnnualRevenue xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey">80000</p1:AnnualRevenue>	<p1:NumberEmployees xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey">14</p1:NumberEmployees>
296	<p1:AnnualRevenue xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey">80000</p1:AnnualRevenue>	<p1:NumberEmployees xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey">15</p1:NumberEmployees>

The **value()** Method

The **value()** method is useful for extracting scalar values from XML documents as a relational value. This method takes an XQuery expression that identifies a single node, and the desired SQL type to be returned. The value of the XML node is returned cast to the specified SQL type.

Example of a **value()** method

Using the previous example as a starting point, the **value** method can be used to return the contained values as SQL Types.

- Returns the specified SQL type
- Returns the data as a relational column
- There are some SQL types that can't be returned:
 - xml data type, (CLR) user-defined type, image, text, ntext, or sql_variant
- Example:


```
SELECT Demographics.value(
        '/StoreSurvey/NumberEmployees[1]','int'
      ) AS Staff
      FROM sales.store;
```

value() Method

```
WITH XMLNAMESPACES ( DEFAULT 'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey' )
SELECT Top 3 BusinessEntityID,
       Demographics.value('/StoreSurvey/AnnualRevenue/text())[1]','decimal') AS Revenue,
       Demographics.value('/StoreSurvey/NumberEmployees/text())[1]','int') As Staff
FROM   Sales.Store;
```

The previous Transact-SQL makes a few amendments to the XQuery, the main one being the addition of **[1]** to return the first element to the **value()** method. The **value** method takes a second parameter that is the SQL type that needs to be returned. There are some exclusions on the SQL type that can be returned. The types not allowed are the **xml** data type, a common language runtime (CLR) user-defined type, **image**,

text, ntext, or sql_variant data type. In the previous example, the **value()** method is returning a decimal value for the revenue, and an integer for the number of staff. The result of executing this code is:

BusinessEntityID	Revenue	Staff
292	80000	13
294	80000	14
296	80000	15

The exist() Method

Use the **exist()** method to check for the existence of a specified value. The **exist()** method enables the user to perform checks on XML documents to determine whether the result of an XQuery expression is empty or nonempty. The result of this method is:

- 1, if the XQuery expression returns a nonempty result.
- 0, if the result is empty.
- NULL, if the XML instance itself is NULL.

- Checks for the existence of a specific value
- Should be used in preference of a value() method for better performance
- Returns three possible values:
 - 1, indicates that the value was found
 - 0, indicates that the value wasn't found
 - NULL, indicates the XML being inspected in NULL
- Example:

```
SELECT BusinessEntityID
FROM Sales.Store
WHERE Demographics.exist(
  '/StoreSurvey[NumberEmployees=14]'
```

Whenever possible, use the **exist()** method on the xml data type instead of the **value()** method. The **exist()** method will perform better and is most helpful when used in a SQL WHERE clause. The query will utilize XML indexes more effectively than the **value()** method.

Example of an exist() Method

Amend the Transact-SQL code to return only the stores that have a specific number of employees.

exist() Method

```
WITH XMLNAMESPACES ( DEFAULT 'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/StoreSurvey' )
SELECT TOP 3 BusinessEntityID,
       Demographics.value('(/StoreSurvey/AnnualRevenue)[1]', 'decimal') AS Revenue,
       Demographics.value('(/StoreSurvey/NumberEmployees)[1]', 'int') As Staff
FROM Sales.Store
WHERE Demographics.exist('/StoreSurvey[NumberEmployees=14]') = 1;
```

The previous example will return all the stores in the Sales.Store table that have exactly 14 members of staff. The WHERE clause can make use of valid XQuery expression. Running the above will result in the following:

BusinessEntityID	Revenue	Staff
294	80000	14
344	80000	14
372	80000	14

The modify() Method

You can perform data manipulation operations on an XML instance by using the **modify()** method. The **modify()** method changes the contents of an XML document.

You can use the **modify()** method to alter the content of an xml type variable or column. This method takes an XML data manipulation language (DML) statement to insert, update, or delete nodes from the XML data. You can only use the **modify()** method of the xml data type in the SET clause of an UPDATE statement. You can insert, delete, and update one or more nodes by using the insert, delete, and replace value of keywords, respectively.

Note that, unlike the previous methods, an error is returned if NULL is passed to the **modify()** method.

- The **modify()** method:
 - Modifies the contents of XML, in either an XML variable or column
 - Depending on the options selected, modify can:
 - Insert a new XML element—via **insert** command
 - Update the contents on an existing element—via the **replace value of** command
 - Delete an XML element—via **delete** command
 - If updating an XML column, the **modify()** method has to be used in an UPDATE statement

Example of an Insert modify() Method

In the following example, all stores that have a staff level of 14 have an extra <Comments> element added after NumberEmployees:

Insert modify() Method

```
WITH XMLNAMESPACES ( DEFAULT 'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/StoreSurvey' )
UPDATE Sales.Store
SET Demographics.modify(
  insert(<Comments>Problem with staff levels</Comments>)
  after(/StoreSurvey/NumberEmployees)[1]
')
WHERE Demographics.exist('/StoreSurvey[NumberEmployees=14]') = 1;
```

The previous example will change all the rows to have the following XML structure; note the added comments element at the bottom of the XML document:

```
<StoreSurvey xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/StoreSurvey">
  <AnnualSales>800000</AnnualSales>
  <AnnualRevenue>80000</AnnualRevenue>
  <BankName>International Bank</BankName>
  <BusinessType>BM</BusinessType>
  <YearOpened>1991</YearOpened>
  <Specialty>Touring</Specialty>
  <SquareFeet>18000</SquareFeet>
  <Brands>4+</Brands>
  <Internet>T1</Internet>
  <NumberEmployees>14</NumberEmployees>
  <p1:Comments xmlns:p1="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
  works/StoreSurvey">Problem with staff levels</p1:Comments>
</StoreSurvey>
```

Example of a Delete modify() Method

This example Transact-SQL will delete the previously inserted comment elements from the XML column Demographics:

Delete modify() Method

```
WITH XMLNAMESPACES ( DEFAULT 'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/StoreSurvey' )
UPDATE Sales.Store
SET Demographics.modify(
    delete (/StoreSurvey/Comments)[1]
)
WHERE Demographics.exist('/StoreSurvey[NumberEmployees=14]') = 1;
```

Example of a Replace modify() Method

In this example, the banking operation for any store where the number of employees is greater than 99 needs to be changed to a different bank.

Replace modify() Method

```
WITH XMLNAMESPACES ( DEFAULT 'http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/StoreSurvey' )
UPDATE Sales.Store
SET Demographics.modify(
    replace value of (/StoreSurvey/BankName)[1]
    with "High Value Stores Banking"
)
WHERE Demographics.exist('/StoreSurvey[NumberEmployees>99]') = 1;
```

Demonstration: XQuery Methods in a DDL Trigger

In this demonstration, you will see how to use XQuery in DDL triggers.

Demonstration Steps

1. Ensure that you have completed the previous demonstration.
2. In SSMS, in Solution Explorer, double-click **Demonstration 5.sql**.
3. Select the code under **Step 1**, and then click **Execute**.
4. Select the code under **Step 2**, and then click **Execute** to create the trigger.
5. Select the code under **Step 3**, and then click **Execute** to test the trigger.
6. Select the code under **Step 4**, and then click **Execute** to drop the trigger.
7. Select the code under **Step 5**, and then click **Execute** to create a trigger to enforce naming conventions.
8. Select the code under **Step 6**, and then click **Execute** to test the trigger. Note that the code to create a stored procedure named **sp_GetVersion** fails, due to the trigger.
9. Select the code under **Step 7**, and then click **Execute** to create a trigger to enforce tables to have primary keys.
10. Select the code under **Step 8**, and then click **Execute** to test the trigger. Note that the CREATE TABLE statement will fail because there is no primary key defined.
11. Select the code under **Step 9**, and then click **Execute** to clean up the database.
12. Leave SSMS open for the next demonstration.

Categorize Activity

Categorize each item against its correct XQuery method. Indicate your answer by writing the method number to the right of each item.

Items	
1	Returns a scalar SQL Type.
2	Has insert, delete and replace value of as options.
3	Returns 1, 0 or NULL.
4	Need to specify [1] to select a single XML element.
5	Normally used in an UPDATE statement.
6	Should be used in preference to a value() method.
7	Can be used in the SELECT or WHERE clause.
8	Normally used in the WHERE clause.

Category 1	Category 2	Category 3
value() Method	modify() Method	exist() Method

Lesson 6

Shredding XML

Another scenario is the need to extract relational data from an XML document. For example, you might receive a purchase order from a customer in XML format. You then parse the XML to retrieve the details of the items that you need to supply.

The extraction of relational data from within XML documents is referred to as "shredding" the XML documents. There are two ways to do this. SQL Server 2000 introduced the creation of an in-memory tree that you could then query by using an OPENXML function. Although that is still supported, SQL Server 2005 introduced the XQuery **nodes()** method; in many cases, this will be an easier way to shred XML data.

In addition to covering these areas in this lesson, you will also see how Transact-SQL provides a way of simplifying how namespaces are referred to in queries.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how to shred XML data.
- Use system stored procedures for creating and managing in-memory node trees that have been extracted from XML documents.
- Use the OPENXML function.
- Work with XML namespaces.
- Use the **nodes()** method.

Overview of Shredding XML Data

There are two approaches for shredding XML data.

The first is to query an in-memory tree that represents the XML. You can use the **sp_xml_preparedocument** system stored procedure to create an in-memory node tree from an XML document that will make querying the XML data possible. You can then obtain relational data from within an XML document.

Shredding XML with OPENXML

Steps for shredding XML with OPENXML could be:

1. Receive an XML document.
2. Call **sp_xml_preparedocument** to create an in-memory node tree, based on the input XML.
3. Use the OPENXML table-valued function to query the in-memory node tree and extract the relational data.
4. Process the retrieved relational data with other relational data as part of standard Transact-SQL queries.
5. Call **sp_xml_removedocument** removes the node tree from memory.

- There are two approaches to shredding XML
 - Using OPENXML and associated stored procedures
 - Using the `xml.nodes()` method
- In most situations, `nodes()` method will perform better than OPENXML

Shredding XML with the nodes() Method

Using the **nodes()** xml data method is the second approach to shredding and will be typically used alongside other XQuery methods, such as **value()** and **query()** to extract the required data.

For example:

nodes() Method

```
WITH XMLNAMESPACES ('http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/Resume' AS ns)
SELECT candidate.JobCandidateID,
       Employer.value('ns:Emp.StartDate','date') As StartDate,
       Employer.value('ns:Emp.EndDate','date') EndDate,
       Employer.value('ns:Emp.OrgName','nvarchar(4000)') As CompanyName
  FROM HumanResources.JobCandidate AS candidate
 CROSS APPLY candidate.Resume.nodes('/ns:Resume/ns:Employment') AS Resume(Employer)
 WHERE JobCandidateID IN (1,2,3);
```

The previous example loops through the resumés of candidates in the HumanResources.JobCandidate table. It uses the **nodes()** method to select the ns:Employment XML element and obtain information about the candidates employers. Executing the previous example returns the following results:

JobCandidateID	StartDate	EndDate	CompanyName
1	2000-06-01	2002-09-30	Wingtip Toys
1	1996-11-15	2000-05-01	Blue Yonder Airlines
1	1994-06-10	1996-07-22	City Power and Light
2	1994-06-15	NULL	Wingtip Toys
3	1998-08-31	2002-12-28	Trey Research
3	1995-06-15	1998-08-01	Contoso Pharmaceuticals
3	1993-05-10	1995-06-01	Southridge Video

Both methods of shredding will be discussed further in the next topics.

Stored Procedures for Managing In-Memory Node Trees

Before you can use the OPENXML functionality to navigate XML documents, you should create an in-memory node tree. This is done by using the **sp_xml_preparedocument** system stored procedure.

sp_xml_preparedocument

sp_xml_preparedocument is a system stored procedure that takes XML either as the untyped xml data type or as XML stored in the nvarchar data type; creates an in-memory node tree from the XML (to make it easier to navigate); and returns a handle to that node tree.

sp_xml_preparedocument reads the XML text that was provided as input, parses the text by using the Microsoft XML Core Services (MSXML) parser (Msxmlsql.dll), and provides the parsed document in a state that is ready for consumption. This parsed document is a tree representation of the various nodes in the XML document, such as elements, attributes, text, and comments.

Before you call **sp_xml_preparedocument**, you need to declare an integer variable to be passed as an output parameter to the procedure call. When the call returns, the variable will then be holding a handle to the node tree.

It is important to realize that the node tree must stay available and unmoved in visible memory because the handle is basically a pointer that needs to remain valid. This means that, on 32-bit systems, the node tree cannot be stored in Address Windowing Extensions (AWE) memory.

sp_xml_removedocument

sp_xml_removedocument is a system stored procedure that frees the memory that a node tree occupies and invalidates the handle.

In SQL Server 2000, **sp_xml_preparedocument** created a node tree that was session-scoped; that is, the node tree remained in memory until the session ended or until **sp_xml_removedocument** was called. A common coding error was to forget to call **sp_xml_removedocument**. Leaving too many node trees to remain in memory was known to cause a severe lack of available low-address memory on 32-bit systems. Therefore, a change was made in SQL Server 2005 that made the node trees created by **sp_xml_preparedocument** become batch-scoped rather than session-scoped. Even though the tree will be removed at the end of the batch, it is good practice to explicitly call **sp_xml_removedocument** to minimize the use of low-address memory as much as possible.

Note that 64-bit systems generally do not have the same memory limitations as 32-bit systems.

- Create an in-memory node tree by using **sp_xml_preparedocument**
 - Uses the Microsoft XML Core Services
 - Delete the created node tree and free the memory by using **sp_xml_removedocument**

OPENXML Function

The OPENXML function provides a rowset over in-memory XML documents, which is similar to a table or a view. OPENXML gives access to the XML data as though it is a relational rowset. It does this by providing a rowset view of the internal representation of an XML document.

After you have created an in-memory node tree of an XML document by using

sp_xml_preparedocument, you can use

OPENXML to write queries against the document.

For example, you might have to extract a list of products that you need to supply to a customer from an XML-based order that the customer sent to you. OPENXML provides a rowset view of the document, based on the parameters that are passed to it.

The parameters that are passed to OPENXML are: the XML document handle; a rowpattern, which is an XPath expression that maps the nodes of XML data to rows; and a flag that indicates whether to use attributes rather than elements by default. Associated with the OPENXML clause is a WITH clause that provides a mapping between the rowset columns and the XML nodes.

Example Using the OPENXML Function

The following example uses OPENXML to shred data from a resumé:

OPENXML Function

```
DECLARE @xmldoc AS int, @xml AS xml;
SELECT @xml=Resume FROM HumanResources.JobCandidate WHERE JobCandidateID=1;
EXEC sp_xml_preparedocument @xmldoc OUTPUT,
    @xml,
    '<root xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/Resume">';
SELECT * FROM OPENXML(@xmldoc, '/ns:Resume/ns:Employment', 2)
WITH (
    [ns:Emp.StartDate] DATETIME
    , [ns:Emp.EndDate] DATETIME
    , [ns:Emp.OrgName] NVARCHAR(1000)
)
EXEC sp_xml_removedocument @xmldoc;
```

In the above example, the OPENXML function is passed an @xml variable that contains a single XML document, representing a resumé from the HumanResources.JobCandidate table. The XPath expression "/ns:Resume/ns:Employment" selects the Employment nodes from the document. Finally, the optional flag of 2 indicates to the OPENXML function that the WHERE clause is matching on elements instead of attributes. Executing the previous Transact-SQL produces these results:

ns:Emp.StartDate	ns:Emp.EndDate	ns:Emp.OrgName
2000-06-01 00:00:00.000	2002-09-30 00:00:00.000	Wingtip Toys
1996-11-15 00:00:00.000	2000-05-01 00:00:00.000	Blue Yonder Airlines
1994-06-10 00:00:00.000	1996-07-22 00:00:00.000	City Power and Light

The optional flag is a byte value, and therefore can be a combination of the following options:

Byte value	Description
0	Defaults to attribute-centric mapping. This is the default if no value is provided.
1	Use the attribute-centric mapping.
2	Use the element-centric mapping.
8	Can be combined (logical OR) with the previous values. In the context of retrieval, this flag indicates that the consumed data should not be copied to the overflow property @mp:xmltext.

Creating an Edge Table

The OPENXML function, along with shredding data, can also be used to create an edge table. An edge table is a relational table view of the XML document. Each XML entity equates to a row in the edge table. You can think of each row in the edge table as a node in the logical representation of the XML document. Each row of the edge table includes a set of columns that describe the entity.

To create an edge table, simply call OPENXML without a WHERE clause.

Create an Edge Table

```
DECLARE @xmldoc AS int, @xml AS xml;
SELECT @xml=Resume FROM HumanResources.JobCandidate WHERE JobCandidateID=1;
EXEC sp_xml_preparedocument @xmldoc OUTPUT,
    @xml,
    '<root xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
works/Resume"/>';
SELECT * FROM OPENXML(@xmldoc, '/ns:Resume/ns:Employment', 2);
EXEC sp_xml_removedocument @xmldoc;
```

The first few rows returned from executing the preceding code are:

id	parentid	nodetype	localname	prefix	namespaceuri	datatype	prev	text
10	0	1	Employment	ns	http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume	NULL	9	NUL L
11	10	1	Emp.StartDate	ns	http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume	NULL	NULL	NUL L
86	11	3	#text	NULL	NULL	NULL	NULL	200 0- 06- 01Z
12	10	1	Emp.EndDate	ns	http://schemas.microsoft.com/sqlserver/2004/0	NULL	11	NUL L

id	parentid	nodetype	localname	prefix	namespaceuri	datatype	prev	text
					7/adventure-works/Resume			
87	12	3	#text	NULL	NULL	NULL	NULL	2002-09-30Z
13	10	1	Emp.OrgName	ns	http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume	NULL	12	NUL L
88	13	3	#text	NULL	NULL	NULL	NULL	Win gtip Toys
14	10	1	Emp.JobTitle	ns	http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume	NULL	13	NUL L
89	14	3	#text	NULL	NULL	NULL	NULL	Lea d Mac hinis t

Working with XML Namespaces

Earlier in this module, you saw how an XML namespace is a collection of names that you can use as element or attribute names in an XML document. The namespace qualifies names uniquely to avoid naming conflicts with other elements that have the same name.

Namespaces can be specified in several ways

When being used with the **sp_xml_preparedocument**, the namespace is declared as the last parameter:

- **sp_xml_preparedocument** accepts namespaces as the last parameter
- Use namespace prefix in all XPath expressions
- Namespaces can be used with FOR XML statements and xml data methods with a WHERE clause

```
EXEC sp_xml_preparedocument @xmldoc OUTPUT, @xml,
'<root xmlns:ns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/Resume"/>';
```

Namespaces can be used in one of two ways when working with the xml data methods and FOR XML statements. The first way requires that they are repeated for every method call where they are required. Taking a previous **query()** example and rewriting it:

```
SELECT Top 3 BusinessEntityID,
       Demographics.query(
           declare default element namespace
           "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey";
           /StoreSurvey/AnnualRevenue
       ') AS Revenue,
       Demographics.query(
           declare default element namespace
           "http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey";
           /StoreSurvey/NumberEmployees
       ') AS Staff
  FROM Sales.Store;
```

The preferred method for referencing an XML namespace is to use the WITH statement. The benefits are that it only has to be declared once at the beginning of the query. The WITH statement can be used for both FOR XML statements and xml data method calls. For example:

```
WITH XMLNAMESPACES (DEFAULT
    'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/StoreSurvey')
SELECT   Top 3 BusinessEntityID,
         Demographics.query('/StoreSurvey/AnnualRevenue') AS Revenue,
         Demographics.query('/StoreSurvey/NumberEmployees') AS Staff
    FROM     Sales.Store;
```

nodes() Method

The **nodes()** method provides an easier way to shred XML into relational data than OPENXML and its associated system stored procedures.

nodes() Method

The **nodes()** method is an XQuery method that is useful when you want to shred an xml data type instance into relational data. It is a table-valued function that enables you to identify nodes that will be mapped into a new relational data row.

Every xml data type instance has an implicitly provided context node. For the XML instance that is stored in a column or a variable, this is the document node. The document node is the implicit node at the top of every xml data type instance.

The result of the **nodes()** method is a rowset that contains logical copies of the original XML instances. In these logical copies, the context node of every row instance is set to one of the nodes that is identified with the query expression. This means that subsequent queries can navigate, relative to these context nodes.

You should be careful about the query plans that are generated when you use the **nodes()** method. In particular, no cardinality estimates are available when you use this method. This has the potential to lead to poor query plans. In some cases, the cardinality is simply estimated to be a fixed value of 10,000 rows. This might cause an inappropriate query plan to be generated if your XML document contained only a handful of nodes.

- Shreds XML variables into relational data
- Requires the APPLY operator with XML columns
- Example:

```
SELECT person.BusinessEntityID,
       person.LastName,
       contact.value('act:number','nvarchar(4000)') As
       ChangeInContactNumber
  FROM Person.Person AS person
 CROSS APPLY person.AdditionalContactInfo.nodes(
 '/AdditionalContactInfo/act:telephoneNumber'
 ) AS helpdesk(contact)
 WHERE person.AdditionalContactInfo IS NOT NULL;
```

CROSS APPLY and Table-Valued Functions

The **nodes()** method is a table-valued function that is normally called by using the CROSS APPLY or OUTER APPLY operations.

APPLY operations cause table-valued functions to be called for each row in the left table of the query.

The following example searches for any telephone numbers that have been captured by support staff and recorded as additional contact information:

CROSS APPLY and Table-Valued Functions

```
WITH XMLNAMESPACES (
    'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ContactTypes' as act,
    DEFAULT 'http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ContactInfo'
)
SELECT person.BusinessEntityID,
       person.LastName,
       contact.value('act:number', 'nvarchar(4000)') As ChangeInContactNumber
FROM Person.Person AS person
CROSS APPLY
person.AdditionalContactInfo.nodes('/AdditionalContactInfo/act:telephoneNumber') AS
helpdesk(contact)
WHERE person.AdditionalContactInfo IS NOT NULL;
```

In this query, for every row in the Person.Person table, where the AdditionalContactInfo column isn't NULL, the **nodes()** method is called. When table-valued functions are used in queries like this, you must provide an alias for both the derived table and the columns that it contains. In this case, the alias provided to the derived table is **helpdesk**, and the alias provided to the extracted column is **contact**.

One output row is being returned for each node at the level of the XPath expression **/AdditionalContactInfo/act:telephoneNumber**. From the returned XML column (**contact**), the ChangeInContactNumber column is generated by calling the **value()** method. Executing the previous query returns these results:

BusinessEntityID	LastName	ChangeInContactNumber
291	Achong	425-555-1112
293	Abel	206-555-2222
293	Abel	206-555-1234
295	Abercrombie	605-555-9877
303	Smith	206-555-2222
307	Adams	253-555-4689

Note that, in these results, two telephone numbers have been returned for Abel. Looking at the XML in the AdditionalContactInfo row, there are three "act:number" XML nodes:

- 206-555-2222
- 206-555-1234
- 206-555-1244

The last number is not contained in the results because the XPath to it is /AdditionalContactInfo/act:pager instead of /AdditionalContactInfo/act:telephoneNumber.

```

<AdditionalContactInfo
  xmlns="http://schemas.microsoft.com/sqlserver/2004/07/adventure-works/ContactInfo"
  xmlns:crm="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
  works/ContactRecord"
  xmlns:act="http://schemas.microsoft.com/sqlserver/2004/07/adventure-
  works/ContactTypes">
  These are additional phone and pager numbers for the customer.
  <act:telephoneNumber><act:number>206-555-2222</act:number>
  <act:SpecialInstructions>On weekends, contact the manager at this number.
  </act:SpecialInstructions></act:telephoneNumber>
  <act:telephoneNumber><act:number>206-555-1234</act:number> </act:telephoneNumber>
  <act:pager><act:number>206-555-1244</act:number><act:SpecialInstructions>Do not page
  between 9:00 a.m. and 5:00 p.m.</act:SpecialInstructions></act:pager>
  Customer provided this additional home address...

```

Demonstration: Shredding XML

In this demonstration, you will see how to:

- Shred XML data by using the **nodes()** method.
- Shred XML using the OPENXML method.

Demonstration Steps

1. Ensure that you have completed the previous demonstration.
2. In SSMS, in Solution Explorer, double-click **Demonstration 6.sql**.
3. Select the code under **Step 1**, and then click **Execute**.
4. Select the code under **Step 2**, and then click **Execute** to select the contents of the **dbo.DatabaseLog** table.
5. In the results pane, in the **Xmlevent** column, click the first entry to view the format of the XML. Note that this is the EVENTDATA structure returned by the DDL and LOGON triggers.
6. Switch back to the **Demonstration 6.sql** pane, select the code under **Step 4**, and then click **Execute**. Compare the first row in the results with the data shown in the **Xmlevent1.xml** pane.
7. Select the code under **Step 5**, and then click **Execute**.
8. Select the code under **Step 6**, and then click **Execute** to show the same results obtained by using OPENXML.
9. Close SSMS without saving any changes.

Check Your Knowledge

Question
<p>Which of the following statements about the OPENXML statement is false?</p>
<p>Select the correct answer.</p>
<input type="checkbox"/> It can only be used to shred XML.
<input type="checkbox"/> You must call the sp_xml_preparedocument to create an in-memory node tree before using it.
<input type="checkbox"/> You should call the sp_xml_removedocument after using it.
<input type="checkbox"/> In most situations, it will perform worse than the xml.nodes() method.

Lab: Storing and Querying XML Data in SQL Server

Scenario

A new developer in your organization has discovered that SQL Server can store XML directly. He is keen to use this mechanism extensively. In this lab, you will decide on the appropriate usage of XML in the documented application.

You also have an upcoming project that will require the use of XML data in SQL Server. No members of your current team have experience working with XML data in SQL Server. You need to learn how to process XML data within SQL Server and you have been given some sample queries to assist with this learning. Finally, you will use what you have learned to write a stored procedure for the marketing system that returns XML data.

Objectives

After completing this lab, you will be able to:

- Determine appropriate use cases for storing XML in SQL Server.
- Test XML storage in variables.
- Retrieve information about XML schema collections.
- Query SQL Server data as XML.
- Write a stored procedure that returns XML.

Estimated Time: 45 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Determining When to Use XML

Scenario

In this exercise, you should read the list of scenarios that your new developer has provided. Determine which are appropriate for XML storage in SQL Server, and which are not. Write "Yes" or "No" next to each scenario.

Scenarios

Scenario Requirements
Existing XML data that is stored, but not processed.
Storing attributes for a customer.
Relational data that is being passed through a system, but not processed in it.
Storing attributes that are nested (that is, attributes stored within attributes).

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Review the List of Scenario Requirements

► **Task 1: Prepare the Lab Environment**

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab14\Starter** folder as Administrator.

► **Task 2: Review the List of Scenario Requirements**

1. Review the list of requirements.
2. For each requirement, determine whether it is suitable for XML storage.

Results: After completing this exercise, you will have determined the appropriate use cases for XML storage.

Exercise 2: Testing XML Data Storage in Variables

Scenario

In this exercise, you will explore how XML data is stored in variables. From a set of sample XML queries, you will review the effect of executing each query.

The main tasks for this exercise are as follows:

1. Review, Execute, and Review the Results of the XML Queries

► **Task 1: Review, Execute, and Review the Results of the XML Queries**

1. Open SQL Server Management Studio and connect to the MIA-SQL instance of SQL Server using Windows authentication.
2. Open **D:\Labfiles\Lab14\Starter\InvestigateStorage.sql**.
3. For each query in the file, review the code, execute the code, and determine how the output results relate to the queries.
4. Close **InvestigateStorage.sql** without saving any changes.
5. Leave SSMS open for the next exercise.

Results: After this exercise, you will have seen how XML data is stored in variables.

Exercise 3: Using XML Schemas

Scenario

For some of the XML processing that you will perform in your upcoming project, you need to validate XML data by using XML schemas. In SQL Server, XML schemas are stored in XML schema collections. You need to investigate how these schemas are used. You have been given a set of sample queries to assist with this. In this exercise, you will review the effect of executing these queries.

The main tasks for this exercise are as follows:

1. Review, Execute, and Review the Results of the XML Queries

► Task 1: Review, Execute, and Review the Results of the XML Queries

1. Open **D:\Labfiles\Lab14\Starter\XMLSchema.sql**.
2. For each query in the file, review the code, execute the code, and determine how the output results relate to the queries.
3. Leave SSMS open for the next exercise.

Results: After this exercise, you will have seen how to create XML schema collections.

Exercise 4: Using FOR XML Queries

Scenario

For some of the XML processing that you must perform in your upcoming project, you will need to return XML data. You should investigate how to do this. You have been given a set of sample queries to assist with this. In this exercise, you will review the effect of executing these queries.

The main tasks for this exercise are as follows:

1. Review, Execute, and Review the Results of the FOR XML Queries

► Task 1: Review, Execute, and Review the Results of the FOR XML Queries

1. Open **D:\Labfiles\Lab14\Starter\XMLQuery.sql**.
2. For each query in the file, review the code, execute the code, and determine how the output results relate to the queries.
3. Leave SSMS open for the next exercise.

Results: After this exercise, you will have seen how to use FOR XML.

Exercise 5: Creating a Stored Procedure to Return XML

Scenario

A new web service is being added to the marketing system. In this exercise, you need to create a stored procedure that will query data from a table and return it as an XML value.

Supporting Documentation

Stored Procedure Specifications

Stored Procedure	Production.GetAvailableModelsAsXML
Input Parameters:	None.
Output Parameters:	None.
Returned Rows:	One XML document with attribute-centric XML. Root element is AvailableModels. Row element is AvailableModel. Row contains ProductID, ProductName, ListPrice, Color and SellStartDate (from Marketing.Product) and ProductModelID and ProductModel (from Marketing.ProductModel) for rows where there is a SellStartDate but not yet a SellEndDate.
Output Order:	Rows within the XML should be in order of SellStartDate ascending, and then ProductName ascending. That is, sort by SellStartDate first, and then ProductName within SellStartDate.

Stored Procedure	Sales.UpdateSalesTerritoriesByXML
Input Parameters:	@SalespersonMods xml.
Output Parameters:	None.
Returned Rows:	None.
Actions:	Update the SalesTerritoryID column in the Sales.Salesperson table, based on the SalesTerritoryID values extracted from the input parameter.

Incoming XML Object Format

This is an example of the incoming XML:

This is an example of the incoming XML:

Incoming XML Object Format

```
<SalespersonMods>
    <SalespersonMod BusinessEntityID="274">
        <Mods>
            <Mod SalesTerritoryID="3"/>
        </Mods>
    </SalespersonMod>
    <SalespersonMod BusinessEntityID="278">
        <Mods>
            <Mod SalesTerritoryID="4"/>
        </Mods>
    </SalespersonMod>
</SalespersonMods>
```

The main tasks for this exercise are as follows:

1. Review the Requirements
2. Create a Stored Procedure to Retrieve Available Models
3. Test the Stored Procedure
4. If Time Permits: Create a Stored Procedure to Update the Sales Territories Table

► **Task 1: Review the Requirements**

- Review the stored procedure specification for **WebStock.GetAvailableModelsAsXML** in the exercise scenario.

► **Task 2: Create a Stored Procedure to Retrieve Available Models**

- Create and implement the **Production.GetAvailableModelsAsXML** stored procedure based on the specifications that are provided.

► **Task 3: Test the Stored Procedure**

1. Test the stored procedure by executing it.
2. Review the returned data.

► **Task 4: If Time Permits: Create a Stored Procedure to Update the Sales Territories Table**

1. If time permits, implement the **Sales.UpdateSalesTerritoriesByXML** stored procedure.
2. Test the created stored procedure with the example incoming XML.
3. Close SSMS without saving any changes.

Results: After this exercise, you will have a new stored procedure that returns XML in the AdventureWorks database.

Module Review and Takeaways



Best Practice: This module has considered a variety of different aspects of using XML data within SQL Server, including storing XML data, querying XML data, performance and XML indexes, and shredding XML data.

Review Question(s)

Question: Which XML query mode did you use for implementing the WebStock.GetAvailableModelsAsXML stored procedure?

Module 15

Storing and Querying Spatial Data in SQL Server

Contents:

Module Overview	15-1
Lesson 1: Introduction to Spatial Data	15-2
Lesson 2: Working with SQL Server Spatial Data Types	15-7
Lesson 3: Using Spatial Data in Applications	15-15
Lab: Working with SQL Server Spatial Data	15-20
Module Review and Takeaways	15-23

Module Overview

This module describes spatial data and how this data can be implemented within SQL Server®.

Objectives

After completing this module, you will be able to:

- Describe how spatial data can be stored in SQL Server.
- Use basic methods of the GEOMETRY and GEOGRAPHY data types.
- Query databases containing spatial data.

Lesson 1

Introduction to Spatial Data

Many business applications work with addresses or locations, so it is helpful to understand the different spatial data types, and where they are typically used. SQL Server can process both planar and geodetic data. In this lesson, we will also consider how the SQL Server data types relate to industry standard measurement systems.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how spatial data is useful in a wide variety of business applications.
- Describe the different types of spatial data.
- Describe the difference between planar and geodetic data types.
- Explain the relationship between the spatial data support in SQL Server and the industry standards.
- Work with spatial reference identifiers to provide measurement systems.

Target Applications

There is a perception that spatial data is not useful in mainstream applications. However, this perception is invalid: most business applications can benefit from the use of spatial data.

Business Applications

Although mapping provides an interesting visualization in some cases, business applications can make good use of spatial data for more routine tasks. Almost all business applications involve the storage of addresses or locations. Customers or clients have street addresses, mailing addresses, and delivery addresses. The same is true for stores, offices, suppliers, and many other business-related entities.

- There is a perception that spatial applications are separate to mainstream business applications
- Almost every business application can benefit from spatial data and functions
 - Locations of customers, stores, and offices
 - All addresses
 - Intersections and distances
 - Business intelligence applications particularly benefit from spatial visualizations

Business Intelligence Applications

Business intelligence applications make particularly strong use of spatial data. These applications often deal with results that are best visualized rather than being presented as tables of numbers. Spatial capabilities make it possible to provide very rich forms of visualization.

Common Business Questions

Consider a pet accessories supply company that has stores all over the country. They know where their stores are, and they know where their customers live. The owner suspects that the company's customers are not buying from their nearest store, but has no firm facts to back this up.

It could be true that customers really do purchase from their local store and the owner was misled by a small sample of data. Or perhaps customers really do travel to a store other than their local branch because the products they require are not stocked locally.

These sorts of questions are normal in most businesses, and you can answer them quite easily if you process spatial data in a database.

Types of Spatial Data

SQL Server works with vector-based, two-dimensional (2-D) data, but has some storage options for three-dimensional (3-D) values.

Vector vs. Raster Data

You can store spatial data either as a series of line segments that together form an overall shape (vector storage) or as a series of dots or pixels that are formed by dividing a shape into smaller pieces (raster storage).

- Vector versus raster data
 - Vector: a series of line segments
 - Raster: a series of pixels or dots



Vector storage is the method on which spatial data in SQL Server is based. One advantage of vector-based storage is the way that it can scale. Imagine storing the details of a line. You could divide the line into a series of dots that make up the line. However, if you then zoomed in to an image of the line, the individual dots would become visible, along with the gaps between the dots. This is how raster-based storage works. Alternatively, if the line was stored as the coordinates of the start and end points of the line, it would not matter how much you zoomed in or out, the line would still look complete. This is because it would be redrawn at each level of magnification. This is how vector-based storage works.

To store raster data in SQL Server, you could use the varbinary data type, although you would not be able to directly process the data.

2-D, 3-D, and 4-D

You are probably familiar with seeing two-dimensional drawings or maps on paper. A third dimension would represent the elevation of a point on the map. Four-dimensional (4-D) systems usually incorporate changes in a shape over time.

Spatial data in SQL Server is based on 2-D technology. In some of the objects and properties that it provides, spatial data in SQL Server supports the storage and retrieval of 3-D and 4-D values, but it is important to realize that the third and fourth dimensions are ignored during calculations. This means that if you calculate the distance between, say, a point and a building, the calculated distance is the same, regardless of the floor or level in the building where the point is located.

For more information about the various types of spatial data, see Microsoft Docs:

Spatial Data Types Overview

<http://aka.ms/irtf1a>

Planar vs. Geodetic

Planar systems represent the Earth as a flat surface. Geodetic systems represent the Earth more like its actual shape.

- Planar systems represent the Earth as flat
- Geodetic systems (for example GPS) represent the Earth as round

Planar Systems

Before the advent of computer systems, it was very difficult to perform calculations on round models of the Earth. For convenience, mapping tended to be two-dimensional. Most people are familiar with traditional flat maps of the world.

However, as soon as larger distances are involved, flat maps provide a significant distortion, particularly as you move from the center of the map. When most of the standard maps from atlases were first drawn, they were oriented around where the people who were drawing the maps lived. That meant that the least distortion occurred where the people who were using the maps were based.

Geodetic Systems

Geodetic systems represent the Earth as a round shape. Some systems use simple spheres, but in fact the Earth is not spherical. Spatial data in SQL Server offers several systems for representing the shape of the Earth. Most systems model the Earth as an ellipsoid rather than as a sphere.

OGC Object Hierarchy

The Open Geospatial Consortium (OGC) is the industry body that provides specifications for how processing of spatial data should occur in systems that are based on Structured Query Language (SQL).

- Open Geospatial Consortium is the relevant industry body
 - OGC defined an object tree
- SQL Server data types are based on the **geometry** hierarchy

SQL Specification

One of the two data types that SQL Server provides is the **geometry** data type. It conforms to the OGC Simple Features for SQL Specification version 1.1.0, and is used for planar spatial data. In addition to defining how to store the data, the specification details common properties and methods to be applied to the data.

The OGC defines a series of data types that form an object tree. Curved arc support was added in SQL Server 2012.

Extensions

SQL Server also extends the standards in several ways—it provides a round-earth data type called **geography**, along with several additional useful properties and methods.

Methods and properties that are related to the OGC standard have been defined by using an ST prefix (such as STDistance). Those without an ST prefix are Microsoft® extensions to the standard (such as MakeValid).

Spatial Reference Identifiers

SQL Server supports many measurement systems directly. When you specify a spatial data type in SQL Server, you also specify the measurement system used. You do this by associating a spatial reference ID with the data. A spatial reference ID of zero indicates the lack of a measurement system, and is common when there is no need for a specific measurement system.

- Each spatial instance has a spatial reference identifier (SRID)
- SRID corresponds to a spatial reference system that is a way of performing measurements
- SRID 4326 is the WGS84 system (commonly implemented as the GPS system)
- SRID 0 is used when no system is needed (flat earth)
- When two spatial instances are used in a calculation, their SRIDs must match
- EPSG standard is used to define available SRIDs

Spatial Reference Systems

Any model of the Earth is an approximation, but some models are closer to reality than others. SQL Server supports many different Earth models by using a series of spatial reference identifiers (SRIDs). Each SRID defines the shape of the Earth model, the authority that is responsible for maintaining it, the unit of measure that is used, and a multiplier that determines how to convert the unit of measurement to meters.

SRID 4326

The World Geodetic System (WGS) is commonly used in cartography, geodetics, and navigation. The latest standard is WGS 1984 (WGS 84) and is best known to most people through the Global Positioning System (GPS). GPS is often used in navigation systems and uses WGS 84 as its coordinate system.

In spatial data in SQL Server, SRID 4326 provides support for WGS 84.

If you query the list of SRIDs in SQL Server, the entry for SRID 4326 has the following name. This is formally called the Well-Known Text (WKT) that is associated with the ID:

If you query the list of SRIDs in SQL Server, the entry for SRID 4326 has the following name. This is formally called the Well-Known Text (WKT) that is associated with the ID:

WGS 84

```
GEOCS["WGS 84", DATUM["World Geodetic System 1984", ELLIPSOID["WGS 84", 6378137, 298.257223563]], PRIMEM["Greenwich", 0], UNIT["Degree", 0.0174532925199433]]
```

WGS 84 models the Earth as an ellipsoid (you can imagine it as a squashed ellipsoid), with its major radius of 6,378,137 meters at the equator, a flattening of 1/ 98.257223563 (or about 21 kilometers) at the poles, a prime meridian (that is, a starting point for measurement) at Greenwich, and a measurement that is based on degrees. The starting point at Greenwich is specifically based at the Royal Observatory. The units are shown as degrees and the size of a degree is specified in the final value in the definition. Most geographic data today would be represented by SRID 4326.

For more information about spatial reference identifiers, see Microsoft Docs:

Spatial Reference Identifiers (SRIDs)

<http://aka.ms/bd0j1v>

Demonstration: Spatial Reference Systems

In this demonstration, you will see how to:

- View the available special reference systems.

Demonstration Steps

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **AdventureWorks\Student** with the password **Pa55w.rd**.
2. Run **D:\Demofiles\Mod15\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**. When the script completes, press any key.
4. Start SQL Server Manager Studio, and connect to the **MIA-SQL** instance using Windows authentication.
5. On the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, navigate to **D:\Demofiles\Mod15**, click **20762_15.ssmssln**, and then click **Open**.
7. In Solution Explorer, expand **Queries**, and then double-click the **11 - Demonstration 1A.sql** script.
8. Highlight the Transact-SQL under the comment **Step 1 - Switch to the tempdb database**, and click **Execute**.
9. Highlight the Transact-SQL under the comment **Step 2 - Query the sys.spatial_reference_systems system view**, and click **Execute**.
10. Highlight the Transact-SQL under the comment **Step 3 - Drill into the value for srid 4326**, and click **Execute**.
11. Highlight the Transact-SQL under the comment **Step 4 - Query the available measurement systems**, and click **Execute**.
12. On the **File** menu, click **Close**.

Check Your Knowledge

Question	
Which existing SQL Server data type could you use to store, but not directly process, raster data?	
Select the correct answer.	
	varchar
	varbinary
	int
	string

Lesson 2

Working with SQL Server Spatial Data Types

SQL Server supports two spatial data types, **geometry** and **geography**. They are both system common language runtime (CLR) data types. This lesson introduces each of these data types, and shows how to interchange data by using industry-standard formats.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the support that spatial data in SQL Server provides.
- Explain how system CLR types differ from user defined CLR types.
- Use the **geometry** data type.
- Use the **geography** data type.
- Work with standard spatial data formats.
- Use OGC methods and properties on spatial data.
- Use Microsoft extensions to the OGC standard when working with spatial data.

SQL Server Spatial Data

SQL Server provides rich support for spatial data. It provides two data types: the **geometry** data type, which is suited to flat-earth (planar) models, and the **geography** data type, which is suited to round-earth (geodetic) models.

geometry Data Type

The **geometry** data type is the SQL Server implementation of the OGC Geometry data type. It supports most of the methods and properties of the OGC type plus extensions to the OGC type. You use the **geometry** data type when you are modeling flat-earth models such as two-dimensional diagrams. The **geometry** data type offers a coordinate system based on X and Y.

- Data types
 - **geometry** data type (flat-earth; planar)
 - **geography** data type (round-earth; geodetic)
- Bing maps SDK
- SQL Server Reporting Services map control
- OGC and Microsoft extension methods
 - **ST** prefix on OGC-defined methods
 - No prefix on Microsoft extension methods

geography Data Type

The **geography** data type is a Microsoft extension to the OGC standard that is suitable when you are working with round-earth models such as GPS data. The **geography** data type works with a coordinate system based on longitude and latitude.

 **Note:** Note that, although “latitude and longitude” is a commonly used phrase, the geographical community uses the terminology in the reverse order. When you are specifying inputs for **geographic** data in SQL Server, the longitude value precedes the latitude value.

Additional Support

The Microsoft Bing® Maps software development kit (SDK) integrates closely with spatial data in SQL Server. SQL Server Reporting Services includes a map control that you can use to render spatial data and a wizard to help to configure the map control. The map control is available for reports built using Business Intelligence Development Studio or Report Builder.

An application that stores or retrieves spatial data from a database in SQL Server needs to be able to work with that data as a spatial data type. To make this possible, a separate installer (MSI) file is provided as part of the SQL Server Feature Pack, so that client applications can use the spatial data types in SQL Server. Installing the feature pack on client systems causes an application on the client to "rehydrate" a **geography** object that has been read from a SQL Server database into a **SqlGeography** object within .NET managed code.

ST Prefix

An **ST** prefix has been added to the properties and methods that are implementations of the OGC standards. For example, the X and Y coordinates of a **geometry** object are provided by **STX** and **STY** properties, and the distance calculation is provided by the **STDistance** method.

Microsoft extensions to the OGC standards have no prefix added to the name of the methods or properties. You should take care when referring to properties and methods because they are case-sensitive, even on servers configured for case-insensitivity.

System vs. User SQL CLR Types

Initially user-defined CLR data types were limited to data that could be serialized into 8 KB of storage.

SQL Server 2008 introduced larger CLR object types, increasing the limit to 2 GB. SQL Server 2008 also introduced the concept of "system" CLR types and all previous types are now considered "user" types.

There is a column in sys.assemblies that indicates whether an assembly is a system or user assembly. System CLR types are turned on regardless of the CLR-enabled configuration setting. The setting only affects code in user types. Due to this change, the spatial data types will work even when the CLR-enabled configuration setting is off.

For more information about the CLR-enabled configuration setting, see Microsoft Docs:

clr enabled Server Configuration Option

<http://aka.ms/oq8rnt>

- System types are turned on regardless of the "clr enabled" setting
- **geometry**, **geography**, and **hierarchyid** use large CLR object support
- Call properties and methods on CLR objects by using:
 - `Instance.Property`—for example, `NewYork.STArea`
 - `Border`
 - `Instance-Method()`—for example, `Border.MakeValid()`
 - `Type::StaticMethod()`—for example, `geometry::STGeomFromText()`

The **geometry** and **geography** data types are implemented as CLR types by using managed code. They are defined as system CLR types and work even when CLR integration is not switched on at the SQL Server instance level.

You can see the currently installed assemblies, and whether they are user-defined, by executing the following query:

Currently Installed Assemblies

```
SELECT name,
       assembly_id,
       permission_set_desc,
       is_user_defined
  FROM sys.assemblies;
```

Accessing Properties and Methods

You can access a property of an instance of a spatial data type by referring to it as **Instance.Property**.

As an example of this, look at the following code that is accessing the STX property of a variable called @Location:

Accessing Properties

```
SELECT @Location.STX;
```

You can call methods that are defined on the data types (**geometry** and **geography**) rather than on instances (that is, columns or variables) of those types. This is an important distinction.

As an example of this, look at the following code that is calling the GeomFromText method of the **geometry** data type:

Calling Methods Defined on Data Types

```
SELECT @Location = geometry::STGeomFromText('POINT (12 15)',0);
```

 **Note:** Note that you are not calling the method on a column or variable of the **geometry** data type, but on the **geometry** data type itself. In .NET terminology, this refers to this as calling a public static method on the **geometry** class. Note that the methods and properties of the spatial data types are case-sensitive, even on servers that are configured with case-insensitive default collations.

geometry Data Type

The **geometry** data type is used for flat-earth (planar) data storage and calculations, and is implemented as a system CLR type. It provides comprehensive coverage of the OGC standard.

The **geometry** data type is a two-dimensional data type based on an X and Y coordinate system. In the definition of the type, there is provision for Z (elevation) and M (measure) in addition to the X and Y coordinates. You can enter and retrieve the Z and M values in the **geometry** data type, but it ignores these values when it performs calculations.

- Two-dimensional data type
- Has STX and STY properties
- SRID is not relevant, it defaults to zero
- Comprehensive OGC coverage

You can see the input and output of X, Y, Z, and M in the following code:

geometry Data Type

```
DECLARE @Location geometry;
SELECT @Location = geometry::STGeomFromText('POINT (12 15 2 9)',0);
SELECT @Location.STAsText();
SELECT @Location.AsTextZM();
```

The SQL Server **geometry** data type provides comprehensive coverage of the OGC Geometry data type, and has X and Y coordinates represented by **STX** and **STY** properties.

SRID and Geometry

When you are working with geometric data, the measurement system is not directly relevant. For example, the area of a shape that measures 3×2 is still 6, regardless of whether 3 and 2 are in meters or in inches. For this reason, there is no need to specify an SRID when you are working with geometric data. When you are entering data, the SRID value is typically left as zero.

Spatial Results Viewer

When a SQL Server result includes columns of the **geometry** or **geography** data types, a special spatial results viewer is provided for you to visualize the spatial results.

For more information on the **geometry** data type, see Microsoft Docs:

 **geometry (Transact-SQL)**

<http://aka.ms/hl20zm>

geography Data Type

The **geography** data type is implemented as a system CLR type and used for round-earth values, typically involving actual positions or locations on the Earth. The **geography** data type is an extension to the OGC standard.

The **geography** data type is based on a latitude and longitude coordinate system. Latitude and longitude values are represented by **Lat** and **Long** properties. Unlike the **geometry** data type, where the X and Y coordinates can be any valid number, the **Lat** and **Long** properties must relate to valid latitudes and longitudes for the selected spatial reference system. SRID 4326 (or WGS 84) is the most commonly used spatial reference system for working with the **geography** data type. The **geography** data type can also store, but not process, Z and M values.

- Two-dimensional data type
- Has **Long** and **Lat** properties
- Order is important for polygons

`SELECT Border FROM dbo.Countries WHERE CountryName = 'Italy';`



Result Size Limitations

The implementation of the **geography** data type in SQL Server 2008 required any geography value to be contained within a single hemisphere. This did not mean any specific hemisphere, such as the northern or southern hemispheres, but just that no two points could be more than half the Earth apart if they were contained in the same instance of the geography data type. This limitation was removed in SQL Server 2012.

Point Order in Polygons

When you are defining the shape of a polygon by using a series of points, the order in which the points are provided is significant. Imagine the set of points that define a postal code region. The same set of points actually defines two regions: all of the points inside the postal code region, and all of the points outside the postal code region.

To enclose points, they should be listed in counterclockwise order. As you draw a shape, points to the left of the line that you draw will be enclosed by the shape. The points on the line are also included.

If you draw a postal code region in a clockwise direction, you are defining all points outside the region. In versions of SQL Server before 2012, this would have resulted in an error because results were not permitted to span more than a single hemisphere.

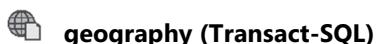
Spatial Results Viewer

A spatial results viewer is provided whenever a result set is displayed in SQL Server Management Studio, with results that include either geometry or geography data.

For geography, the viewer is quite configurable. You can set which column to display, the geographic projection to use for display, such as Mercator or Bonne, and you can choose to display another column as a label over the relevant displayed region.

The spatial results viewer in SQL Server Management Studio is limited to displaying the first 5,000 objects from the result set.

For more details on the geography data type, see Microsoft Docs:



<http://aka.ms/f4ys44>

Spatial Data Formats

In most cases, the internal binary format of any CLR data type is not directly used for input and output of the data type. You have to accommodate string-based representations of the data.

CLR data types (including the **geometry** and **geography** system CLR data types) are stored in a binary format that the designer of the data type determines. Although it is possible to both enter values and generate output for instances of the data type by using a binary string, this is not typically very helpful—you would have to have a detailed understanding of the internal binary format.

- Internal binary format of the spatial types is not normally used directly
- Must be able to enter data and return results as strings
- Parsing
 - Well-known text (WKT)
 - Well-known binary (WKB)
 - Geography markup language (GML) an XML variant
 - **Parse()** assumes WKT
- Output
 - Options to generate output for the above formats including Z and M values
 - **ToString()** provides WKT

The OGC and other organizations that work with spatial data define several formats that you can use for interchanging spatial data. Some of the formats that SQL Server supports are:

- **Well-Known Text (WKT)**. This is the most common string format and is readable by humans.
- **Well-Known Binary (WKB)**. This is a more compact binary representation that is useful for interchange between computers.
- **Geography Markup Language (GML)**. This is the XML-based representation for spatial data.

All CLR data types must implement two string-related methods. The **Parse** method is used to convert a string to the data type and the **ToString** method is used to convert the data type back to a string. Both of these methods are implemented in the spatial types and both assume a WKT format.

Several variations of these methods are used for input and output. For example, the **STAsText** method provides a specific WKT format as output and the **AsTextZM** method is a Microsoft extension that provides the Z and M values, in addition to the two-dimensional coordinates.

For more information on the geometry data type, see Microsoft Docs:

geometry (Transact-SQL)

<http://aka.ms/Oul99w>

OGC Methods and Properties

A wide variety of OGC methods and properties are provided in SQL Server, along with a number of OGC-defined collections. Several of the common methods and properties are described here, but many more exist.

Common Methods

Common OGC methods include:

- The **STDistance** method, which returns the distance between two spatial objects. Note that this does not only apply to points. You can also calculate the distance between two polygons. The result is returned as the minimum distance between any two points on the polygons.
- The **STIntersects** method, which returns 1 when two objects intersect and otherwise returns 0.
- The **STArea** method, which returns the total surface area of a geometry instance.
- The **STLength** method, which returns the total length of the objects in a geometry instance. For example, for a polygon, **STLength** returns the total length of all line segments that make up the polygon.
- The **STUnion** method, which returns a new object that is formed by uniting all points from two objects.
- The **STBuffer** method, which returns an object whose points are within a certain distance of an instance of a **geometry** object.

- Common methods
 - STDistance: the distance between shapes
 - STIntersects: the intersection of two shapes
 - STArea: the area of a shape
 - STLength: the length of a shape
 - STUnion: the shape formed by uniting two shapes
 - STBuffer: the shape formed by points around a shape
 - Common collection properties
 - STPointN: returns a specific point in a collection of points
 - STGeometry: returns a specific geometric shape from a collection of geometries

Common Collection Properties

SQL Server provides support for several collections defined in the OGC specifications. It is possible to hold a **geometry** data type in a **GeometryCollection** object and it can contain several other nested geometry objects. Properties such as **STPointN** and **STGeometryN** provide access to the members of these collections.

For more information on OGC methods and properties on geometry instances, see Microsoft Docs:

OGC Methods on Geometry Instances

<http://aka.ms/Fvhxc>

Microsoft Extensions

In addition to the OGC properties and methods, Microsoft has provided several useful extensions to the standards. Several of these extensions are described in this topic, but many more exist.

Common Extensions

Although the coverage that the OGC specifications provide is good, Microsoft has enhanced the data types by adding properties and methods that extend the standards. Note that the extended methods and properties do not have the **ST** prefix.

- Microsoft has provided a number of extensions to the OGC-defined methods and properties
- Common extensions include:
 - **MakeValid**: returns a valid shape from a potentially invalid shape
 - **Reduce**: reduces the complexity of a shape without changing its basic shape
 - **IsNull**: returns 1 if an object is NULL
 - **AsGML**: returns the object coded as GML
 - **BufferWithTolerance**: returns a buffer around an object, but uses a tolerance value to allow for rounding errors

The **MakeValid** method takes an arbitrary shape and returns another shape that is valid for storage in a **geometry** data type. SQL Server produces only valid geometry instances, but you can store and retrieve invalid instances. You can retrieve a valid instance that represents the same point set of any invalid instance by using the **MakeValid** method.

You can use the **Reduce** method to reduce the complexity of an object while attempting to maintain the overall shape of the object.

The **IsNull** method returns 1 if an instance of a spatial type is NULL; otherwise it returns 0.

The **AsGML** method returns the object encoded as GML.

An example of GML is shown here:

GML

```
<Point xmlns="http://www.opengis.net/gml">
  <pos>12 15</pos>
</Point>
```

GML is excellent for information interchange but the representation of objects in XML can quickly become very large.

The **BufferWithTolerance** method returns a buffer around an object, but uses a tolerance value to allow for minor rounding errors.

For more information about Microsoft extensions, see Microsoft Docs:

geometry Data Type Method Reference

<http://aka.ms/tfcabe>

Demonstration: Spatial Data Types

In this demonstration, you will see how to:

- Work with spatial data types in SQL Server.

Demonstration Steps

1. In SQL Server Manager, in Solution Explorer, under **Queries**, double-click the **21 - Demonstration 2A.sql** script file.
2. Highlight the Transact-SQL under the comment **Step 1 - Switch to the AdventureWorks database**, and click **Execute**.
3. Highlight the Transact-SQL under the comment **Step 2 - Draw a shape using geometry**, and click **Execute**.
4. Click the **Spatial results** tab.
5. Highlight the Transact-SQL under the comment **Step 3 - Draw two shapes**, and click **Execute**.
6. Click the **Spatial results** tab.
7. Highlight the Transact-SQL under the comment **Step 4 - Show what happens if you perform a UNION rather than a UNION ALL. This will fail, as spatial types are not comparable**, and click **Execute**. Note the error message.
8. Highlight the Transact-SQL under the comment **Step 5 - Join the two shapes together**, and click **Execute**.
9. Click the **Spatial results** tab.
10. Highlight the Transact-SQL under the comment **Step 6 - How far is it from New York to Los Angeles in meters?** and click **Execute**.
11. Highlight the Transact-SQL under the comment **Step 7 - Draw the Pentagon**, and click **Execute**.
12. Click the **Spatial results** tab.
13. Highlight the Transact-SQL under the comment **Step 8 - Call the ToString method to observe the use of the Z and M values that are stored but not processed**, and click **Execute**.
14. Highlight the Transact-SQL under the comment **Step 9 - Use GML for input**, and click **Execute**.
15. Click the **Spatial results** tab.
16. Highlight the Transact-SQL under the comment **Step 10 - Output GML from a location (start and end points of the Panama Canal only – not the full shape)**, and click **Execute**.
17. Click the **Spatial results** tab.
18. Highlight the Transact-SQL under the comment **Step 11 - Show how collections can include different types of objects**, and click **Execute**.
19. Click the **Spatial results** tab.
20. On the **File** menu, click **Close**.

Question: You have used a web service to calculate the coordinates of an address. What is this process commonly called, and what services are available?

Lesson 3

Using Spatial Data in Applications

Having learned how spatial data is stored and accessed in SQL Server, you now have to understand the implementation issues that can arise when you are building applications that use spatial data.

Lesson Objectives

After completing this lesson, you will be able to:

- Understand the performance issues with spatial queries.
- Describe the different types of spatial indexes.
- Explain the basic tessellation process used within spatial indexes in SQL Server.
- Implement spatial indexes.
- Explain which geometry methods can benefit from spatial indexes.
- Explain which geography methods can benefit from spatial indexes.
- Describe options for extending spatial data support in SQL Server.

Performance Issues in Spatial Queries

Spatial queries can often involve a large number of data points. Executing methods such as STIntersects for a large number of points is slow. Spatial indexes help to avoid unnecessary calculations, reducing the need for unnecessary complex geometric calculations.

- Spatial queries can involve a large number of data points, causing performance problems

Spatial Indexes

Spatial indexes in SQL Server are based on b-tree structures, but unlike standard relational indexes, which directly locate the specific rows required to answer a query, spatial indexes work in a two-phase manner.

The first phase, known as the primary filter, obtains a list of rows that are of interest. The returned rows are referred to as candidate rows and may include false positives; that is, rows that are not required to answer the query.

In the second phase, a secondary filter checks each individual candidate row to locate the exact rows required to answer the query. The secondary filter executes methods in the WHERE clause of the query on the filtered set of candidate rows, greatly reducing the number of calculations that SQL Server has to make, providing the spatial index has been effective.

- Spatial indexes in SQL Server work in a two-phase manner
- Primary filter
 - Finds all possible candidate rows
 - Can include false positives
- Secondary filter
 - Removes false positives
- **Filter** method shows effectiveness of the primary filter

You can check the effectiveness of a primary filter in SQL Server using the Filter method. The Filter method only applies the primary filter, so you can compare the number of rows that the Filter method returns to the total number of rows.

Tessellation Process

SQL Server spatial indexes use tessellation to minimize the number of calculations that have to be performed. The tessellation process quickly reduces the overall number of rows to a list that might potentially be of interest.

Tessellation Process

SQL Server breaks the problem space into relevant areas by using a four-level hierarchical grid. Each object is broken down and fitted into the grid hierarchy based on which cells it touches.

- Spatial indexes use a tessellation process to produce a list of rows which are of interest
- Four-level hierarchical grid used for breaking problems down
- Three tessellation rules applied
 - Covering rule
 - Cells-per-object rule
 - Deepest cell rule
- Two tessellation schemes depending on data type

Three rules are applied recursively on each grid level to set the depth of the tessellation process, and decide which cells to record in the index. The three rules are:

- **Covering rule**—any cell covered completely by an object is a covered cell and is not tessellated.
- **Cells-per-object rule**—sets the maximum number of cells that can be counted for any object.
- **Deepest-cell rule**—records the bottom most tessellated cells for an object.

Tessellation Scheme

SQL Server uses a different tessellation scheme depending on the data type of the column. Geometry grid tessellation is used for columns of the **geometry** data type and Geography grid tessellation is used for columns of the **geography** data type. The view sys.spatial_index_tessellations returns the tessellation rules of a spatial index.

For more information on the spatial index tessellation process, see Microsoft Docs:



Spatial Indexes Overview

<http://aka.ms/rs5bjy>

Implementing Spatial Indexes

You execute the **CREATE SPATIAL INDEX** statement, providing a name for the index, the table on which the index is to be created, and the spatial data column that needs to be indexed. The table must have a clustered primary key before you can build a spatial index on it; indexes on the **geometry** data type should specify a BOUNDING_BOX setting.

- Created using **CREATE SPATIAL INDEX** statement
- **geometry** has a BOUNDING_BOX setting
- ONLINE index builds are not supported
- A single column can have more than one spatial index
- The table must have a clustered primary key

Index Bounds

Unlike traditional types of index, a spatial index is most useful when it knows the overall area that the spatial data covers. Spatial indexes that are created on the **geography** data type do not have to specify a bounding box because the Earth itself naturally limits the data type.

Spatial indexes on the **geometry** data type specify a BOUNDING_BOX setting. This provides the coordinates of a rectangle that would contain all possible points or shapes of interest to the index. The **geometry** data type has no natural boundaries so, by specifying a bounding box, SQL Server can produce a more useful index. If values arise outside the bounding box coordinates, the primary filter would have to return the rows in which they are contained.

Grid Density

With SQL Server, you can specify grid densities when you are creating spatial indexes. You can specify a value for the number of cells in each grid for each grid level in the index:

- A value of LOW indicates 16 cells in each grid or a 4×4 cell grid.
- A value of MEDIUM indicates 64 cells in each grid or an 8×8 cell grid.
- A value of HIGH indicates 256 cells in each grid or a 16×16 cell grid.

Spatial indexes differ from other types of index because it might make sense to create multiple spatial indexes on the same table and column. Indexes that have one set of grid densities might be more useful than a similar index that has a different set of grid densities for locating data in a specific query.

To make spatial indexes easier to configure, SQL Server has automatic grid density and level selections: GEOMETRY_AUTO_GRID and GEOGRAPHY_AUTO_GRID. The automated grid configuration defaults to an eight-level grid.

Limitations

Spatial indexes do not support the use of ONLINE build operations, which are available for other types of index in SQL Server Enterprise.

For more information on creating spatial indexes, see Microsoft Docs:



CREATE SPATIAL INDEX (Transact-SQL)

<http://aka.ms/xh7lj0>

geometry Methods Supported by Spatial Indexes

Not all geometry methods and predicate forms can benefit from the presence of spatial indexes. The following list details predicates that can potentially make use of a spatial index as a primary filter:

- `geometry1.STContains(geometry2) = 1`
- `geometry1.STDistance(geometry2) < number`
- `geometry1.STDistance(geometry2) <= number`
- `geometry1.STIntersects(geometry2)= 1`
- `geometry1.STOverlaps(geometry2) = 1`
- `geometry1.STTouches(geometry2) = 1`
- `geometry1.STWithin(geometry2)= 1`

- Not all geometry methods benefit from spatial indexes
- Not all predicate forms benefit from spatial indexes

If the predicate in your query is not in one of these forms, spatial indexes that you create will be ignored, potentially resulting in slower queries.

For more information on geometry methods supported by spatial indexes, see MSDN:

Geometry Methods Supported by Spatial Indexes

<http://aka.ms/klss7k>

geography Methods Supported by Spatial Indexes

In a similar way to the **geometry** data type, not all geography methods and predicate forms can benefit from the presence of spatial indexes. The following list shows the specific predicates that can potentially make use of a spatial index as a primary filter:

- `geography1.STIntersects(geography2)= 1`
- `geography1.STEquals(geography2)= 1`
- `geography1.STDistance(geography2) < number`
- `geography1.STDistance(geography2) <= number`

- Not all geography methods benefit from spatial indexes
- Not all predicate forms benefit from spatial indexes

Unless the predicate in your query is in one of these forms, spatial indexes that you create will be ignored and query performance might be affected.

For more information on geography methods supported by spatial indexes, see MSDN:

Geography Methods Supported by Spatial Indexes

<http://aka.ms/wv2g76>

Demonstration: Spatial Data in Applications

In this demonstration, you will see how to:

- Use spatial data in SQL Server to solve some business questions.

Demonstration Steps

1. In SQL Server Manager, in Solution Explorer, under **Queries**, double-click the **31 - Demonstration 3A.sql** script file.
2. Highlight the Transact-SQL under the comment **Step 1 - Open a new query window to the AdventureWorks database**, and click **Execute**.
3. Highlight the Transact-SQL under the comment **Step 2 - Which salesperson is closest to New York?** and click **Execute**.
4. Highlight the Transact-SQL under the comment **Step 3 - Which two salespeople live the closest together?** and click **Execute**.

Note: this will take a few minutes to run.

5. Close SQL Server Management Studio without saving any changes.

Question: Where might spatial data prove useful in your organization?

Lab: Working with SQL Server Spatial Data

Scenario

Your organization has recently started to acquire spatial data within its databases. The new Marketing database was designed before the company started to implement spatial data. One of the developers has provided a table of the locations where prospects live, called Marketing.ProspectLocation. A second developer has added columns to it for Latitude and Longitude and geocoded the addresses. You will make some changes to the system to help support the need for spatial data.

Objectives

After completing this lab you will be able to:

- Use the GEOMETRY data type.
- Use the GEOGRAPHY data type.
- View the results of spatial queries in SSMS.

Estimated Time: 45 Minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Become Familiar with the geometry Data Type

Scenario

You have decided to learn to write queries using the geometry data type in SQL Server. You will review and execute scripts that demonstrate querying techniques.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Review and Execute Queries

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab15\Starter** folder, run **Setup.cmd** as Administrator.

► Task 2: Review and Execute Queries

1. Start SQL Server Manager Studio and connect to the **MIA-SQL** database instance using Windows authentication.
2. Open the SQL Server solution file **20762_15.ssmssln** located in **D:\Demofiles\Mod15\Starter**.
3. Open the query **51 - Lab Exercise 1.sql**.
4. Execute each section of the script individually and review the results.

Exercise 2: Add Spatial Data to an Existing Table

Scenario

In this lab, you have to modify an existing table, Marketing.ProspectLocation, to replace the existing **Latitude** and **Longitude** columns with a new **Location** column of type **geography**. You must migrate the data to the new **Location** column before you delete the existing **Latitude** and **Longitude** columns.

The main tasks for this exercise are as follows:

1. Add a Location Column to the Marketing.ProspectLocation Table
2. Write Code to Assign Values Based on Existing Latitude and Longitude Columns
3. Drop the Existing Latitude and Longitude Columns

► **Task 1: Add a Location Column to the Marketing.ProspectLocation Table**

- Add a **Location** column to the Marketing.ProspectLocation table in the **MarketDev** database.

► **Task 2: Write Code to Assign Values Based on Existing Latitude and Longitude Columns**

- Write code to assign values to the **Location** column, based on the existing **Latitude** and **Longitude** columns.

► **Task 3: Drop the Existing Latitude and Longitude Columns**

- When you are sure that the new column has the correct data, drop the existing **Latitude** and **Longitude** columns.

Results: After this exercise, you should have replaced the existing **Longitude** and **Latitude** columns with a new **Location** column.

Exercise 3: Find Nearby Locations

Scenario

Your salespeople are keen to visit prospects at their own locations, rather than just talk to them on the phone. To minimize effort, they are keen to simultaneously see other prospects in the same area. You will write a stored procedure that provides details of other prospects in the area. To ensure it performs quickly, you will create a spatial index on the table.

The main tasks for this exercise are as follows:

1. Review the Requirements
2. Create a Spatial Index on the Marketing.ProspectLocation Table
3. Design and Implement the Stored Procedure
4. Test the Procedure

► **Task 1: Review the Requirements**

- Read the Requirements.docx located in the folder D:\Labfiles\Lab15\Starter.

► **Task 2: Create a Spatial Index on the Marketing.ProspectLocation Table**

- Create a spatial index on the **Location** column of the Marketing.ProspectLocation table.

► **Task 3: Design and Implement the Stored Procedure**

- Write a stored procedure that will return the details of all prospects within a given distance of a specified prospect. The stored procedure should have input parameters for **ProspectID** and **Distance in kms** and should output customer details in ascending distance order.

► **Task 4: Test the Procedure**

1. Execute the stored procedure to find all prospects within 50 kms of prospectID 2 and check the results.
2. Close SSMS without saving anything.

Results: After completing this lab, you will have created a spatial index and written a stored procedure that will return the prospects within a given distance from a chosen prospect.

Module Review and Takeaways



Best Practice: This module described spatial data and how this data can be implemented within SQL Server.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
Choose the correct data type for your needs.	
Consider the performance of your application.	
Consider security options for tables that hold geography and geometry data types.	

Module 16

Storing and Querying BLOBs and Text Documents in SQL Server

Contents:

Module Overview	16-1
Lesson 1: Considerations for BLOB Data	16-2
Lesson 2: Working with FILESTREAM	16-9
Lesson 3: Using Full-Text Search	16-16
Lab: Storing and Querying BLOBs and Text Documents in SQL Server	16-26
Module Review and Takeaways	16-30

Module Overview

Traditionally, databases have been used to store information in the form of simple values—such as integers, dates, and strings—that contrast with more complex data formats, such as documents, spreadsheets, image files, and video files. As the systems that databases support have become more complex, administrators have found it necessary to integrate this more complex file data with the structured data in database tables. For example, in a product database, it can be helpful to associate a product record with the service manual or instructional videos for that product. SQL Server provides several ways to integrate these files—that are often known as Binary Large Objects (BLOBs)—and enable their content to be indexed and included in search results. In this module, you will learn how to design and optimize a database that includes BLOBs.

Objectives

After completing this module, you will be able to:

- Describe the considerations for designing databases that incorporate BLOB data.
- Describe the benefits and design considerations for using FILESTREAM to store BLOB data on a Windows file system.
- Describe the benefits of using full-text indexing and Semantic Search, and explain how to use these features to search SQL Server data, including unstructured data.

Lesson 1

Considerations for BLOB Data

There are several ways you can store BLOBS and integrate them with the tabular data in a database. Each approach has different advantages and disadvantages. You should consider how a chosen approach affects performance, security, availability, and any other requirements. In this lesson, you will see the principal features of the different approaches.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how BLOBS differ from structured data and list the technical challenges they present.
- Describe how BLOBS can be stored in database files by using columns with the **varchar(max)** data type.
- Describe how database administrators can store BLOBS outside the database and link to them from database tables.
- Describe the FILESTREAM feature and explain the advantages of using it.
- Describe how FileTables extend the FILESTREAM feature and provide access for desktop applications.

What Are BLOBs?

In database terminology, a BLOB is a large amount of unstructured data. Often, but not always, a BLOB represents the contents of a file, such as a document, image, or video. The data in such files is considered to be unstructured because it does not conform to a strict schema of columns and data types like a database table. Although a Word document, for example, has its own internal structure, this is not understood by the database engine and is treated as a simple stream of ones and zeros.

If a BLOB represents a file then its file type, indicated by its file extension, is usually associated with one or more applications for viewing and editing. For example, Word documents with the .docx extension are associated with Microsoft Word. Such applications use Windows APIs to open the file and save changes. They cannot use database APIs to access such files and are not aware of database transactions.

- Technical challenges presented by BLOBS
 - Storage and management costs
 - Security
 - Indexing and searching text data
 - Referential and transactional integrity
- BLOB storage strategies
 - In the database
 - In the file system
 - FILESTREAM
 - FileTable
 - Remote BLOB storage

Technical Challenges Presented by BLOBs

If you integrate BLOBs with your database, the way data is handled changes significantly. For example:

- **Storage and management costs.** If you have many large BLOBs, they can consume large amounts of disk space and backup media space. They also increase backup and restore times.
- **Security.** If BLOBs are stored in the database, the authorization to access them can be controlled by using roles, logins and users, as for the authorization to access tables, views, and so on. However, if BLOBs are stored on the file system, access must be controlled by using Windows accounts, groups, and NTFS permissions.

- **Indexing and searching text data.** BLOBS such as Word files contain large quantities of unstructured text. Users might like to search this text for specific words but standard SQL Server indexes do not support it.
- **Referential and transactional integrity.** You must consider how create, read, update, and delete (CRUD) operations on database rows might affect corresponding BLOBS. For example, if a product is deleted from the product catalog, should the corresponding product manual also be deleted?

BLOB Storage Strategies

In SQL Server, you can choose from the following strategies for BLOB storage and integration:

- **Storing BLOBs in the database.** In this approach, you add a column to a database table with the varbinary(MAX) datatype. The file is stored as binary information within the database file.
- **Storing BLOBs on the file system.** In this approach, you save BLOBs to a shared folder on a file server. You can link files to database rows by adding a column with the varchar() or nvarchar() datatype to the relevant tables, and using it to store a path to the file.
- **FILESTREAM.** If you use the FILESTREAM feature, BLOBs are stored in the file system, although they are accessed through the database as varbinary(MAX) columns. To applications, BLOBs appear to be stored in database tables, but the use of the file system can increase performance and simplify the challenges of referential and transactional integrity.
- **FileTables.** This feature is an extension of FILESTREAM. BLOBs are stored on the file system but can be accessed through the database. In addition, applications can access and modify BLOBs by using the Windows APIs they would use for accessing nondatabase files.
- **Remote BLOB Storage (RBS).** SQL Server RBS is an optional add-on that you can use to place BLOBs outside the database in commodity storage solutions. RBS is extensible and comes with several different providers. Each provider supports a different kind of remote storage solution. Several third-party vendors have created providers for their proprietary storage technologies; you can also develop custom providers.

Storing BLOBs in the Database

In this approach, you create a new column with the data type **varbinary(max)** and use it to store BLOBs within the database itself.

Suitable Data Types

The **varbinary(max)** data type has a maximum size of approximately 2 GB (2,147,483,647 bytes). This constitutes the maximum size of file that you can store within the database. If you expect to store larger files than this limit, you must use another approach to BLOBs.

SQL Server takes a flexible approach to storage for **varbinary(max)** columns. If a BLOB is smaller than 8,000 bytes, it is stored in the same data page as the other columns in the row. This maximizes performance, because the database engine does not have to incur extra page reads to access the BLOB. However, if the BLOB is larger than 8,000 bytes, it is stored in separate data pages with a pointer to those pages stored with the rest of the row. These data pages are in a dedicated binary tree (a b-tree).

- | |
|---|
| <ul style="list-style-type: none"> • Storing BLOBs in the database <ul style="list-style-type: none"> • Varbinary(max)—2 GB max size • 8,000 bytes or less is stored in the same data page • Larger than 8,000 bytes is stored in separate data pages • Use sp_tableoption—large value types out of row • Advantages <ul style="list-style-type: none"> • Everything within the database for backups, restores, transactional and referential integrity, and so on • Disadvantages <ul style="list-style-type: none"> • BLOBs can only be accessed through Transact-SQL • Large BLOBs can reduce performance |
|---|



Note: You can control this storage behavior by using the **large value types out of row** option in the `sp_tableoption` stored procedure. If this option is set to **0**, the behavior is as described above. If the option is set to **1**, then BLOBS are always stored in separate pages, even if they are under the 8,000 byte limit.



Note: Early versions of SQL Server included the **image** data type, which was intended for BLOB storage. This data type is still available in SQL Server but is deprecated and should not be used.

Advantages and Disadvantages

By storing BLOBS in the database, you closely integrate them with associated data stored in other columns. It is possible, for example, for a product manual to be stored with an associated product, because the manual is part of the product's table row. This has the following advantages:

- All data is stored in the database files; there is no requirement to maintain and back up a separate set of folders on the file system where BLOBS are located.
- Restore operations are simplified, because only the database needs to be restored.
- The transactional and referential integrity of BLOB data is automatically maintained by SQL Server.
- Administrators do not need to secure a separate set of folders on the file system.
- Developers writing applications that use your database can access BLOBS by using Transact-SQL and do not have to call separate I/O APIs.
- Full integration with Full-Text Search and Semantic Search for textual data.

However, the following issues may be considered disadvantages of this approach:

- BLOBS can only be accessed through Transact-SQL. Word, for example, cannot open Word documents that are stored directly as database BLOBS.
- Large BLOBS may reduce read performance, because a large number of pages from the b-tree may have to be retrieved from the disk for each BLOB.
- Although restores are simpler, they often take longer, because BLOBS typically add considerable size to database files.

Storing BLOBs on the File System

In this approach, BLOBs are stored outside the database on a dedicated store, such as a shared folder on a file server. You associate a database row with the external BLOB by storing a path to that BLOB.

- Storing BLOBs on the file system
 - BLOBs stored outside the database
 - Store a path to the BLOB
- Two data stores
 - Maintaining data integrity in two places
- Advantages
 - Requires little configuration to store large files
 - Better performance, and reduced fragmentation
- Disadvantages
 - No mechanism for keeping data in step
 - Security must be configured for the file system

For example, a Products table in the database may have a column called "ManualPath" of the **varchar()** data type. In the row for the "Rear Derailleur Shifter", the ManualPath column may store the path "\\DocumentServer\Manuals\RearShifterManual.doc".

 **Note:** In this example, the stored path is a Server Message Block (SMB) path to a file on a file server. Depending on your file store, paths may be in other forms, such as URLs.

Maintaining Two Stores

The challenges presented by this approach arise from the necessity to run two storage locations—the database and the file store.

Atomicity is a major concern. For any operation that alters one of the storage locations, you must consider how the entry in the other location will be affected. For example, if a product is deleted from the catalog, should the corresponding product manual be deleted from the file server? If a product's part number is altered in the database, does this change the need to be propagated to the BLOB—and how should it be updated?

You must also plan how to secure both locations—using logins, users, and GRANTS for the database, and using Windows accounts and permissions (or some other security system) for the file store.

Advantages and Disadvantages

Originally, a database was intended to store tabular structured data. Therefore, removing BLOBS from the database has the following advantages:

- It requires little configuration in the database.
- It avoids database files growing to large volumes. This makes it easier to manage the database. For example, backup and restore operations for the database itself become shorter.
- Read performance for large BLOBS is typically faster than it would be for BLOBS stored in the database.
- BLOBS are less likely to become fragmented on the file system. It is easier to reduce fragmentation on the file system and this can ensure better performance.
- Because BLOBS are stored in a shared folder, applications can access them without going through SQL Server. For example, a user with the correct path can open a manual in Word.

The disadvantages of this approach arise from the less close integration between the BLOBS and their corresponding database rows:

- There is no mechanism for maintaining transactional and referential integrity. For example, if a user moves a BLOB in the file store, the path in the database row will not automatically be updated and will be broken.
- There is another location to back up and restore.
- Security administration must be done twice—once for the database and once for the file store.
- Developers must use two mechanisms to access data—Transact-SQL for access to the database and a Windows API for access to the BLOBS. This adds complexity and increases development time.
- BLOBS are not available for Full-Text Search or Semantic Search.

FILESTREAM

You use the FILESTREAM feature, which was introduced in SQL Server 2008, to store BLOBS on the file system, closely integrated with their corresponding rows. It combines the extra performance you can achieve for large BLOBS served from the file system, with the advantages of storing BLOBS in the database.

FILESTREAM Implementation

FILESTREAM is an attribute of the **varchar(max)** data type. If you enable this attribute on a **varchar(max)** column, SQL Server stores BLOBs in a folder on the NTFS file system. You always access these BLOBs through the database server but you can choose to use either Transact-SQL or Win32 I/O APIs, which have better performance for large files. You can also store BLOBs that have more than the 2 GB size limit for BLOBs stored in the database.

To use FILESTREAM, you must create at least one FILESTREAM filegroup in your database. This is a dedicated kind of filegroup that contains file system directories, called "data containers", instead of the actual BLOBs.

Benefits

When you use FILESTREAM, BLOBs are stored outside the database on the file system, but from the point of view of applications, they appear to be within the database. This has the advantage of high performance using external BLOBs, and close integration when you store BLOBs within the database.

Advantages

- When BLOBs are larger than about 1 MB, read performance will be greater with FILESTREAM than for BLOBs stored within the database.
- BLOBs are fully integrated with the database for management and security.
- SQL Server automatically maintains referential and transactional integrity.
- BLOBs are available for full-text searches and semantic searches.
- There is no upper limit on the size of BLOBs stored in FILESTREAM columns.
- Developers access all BLOBs through the database server and can use either Transact-SQL or Win32 APIs.

Disadvantages

- Applications cannot access BLOBs directly; instead, developers must write code that reads and writes BLOB data.
- BLOBs must be stored on a hard drive installed on the database server itself; you cannot use a shared folder on another file server.



Note: You can use a Storage Area Network (SAN) to store FILESTREAM BLOBs, because these appear as local hard drives to the database server.

- | |
|---|
| <ul style="list-style-type: none"> • FILESTREAM implementation <ul style="list-style-type: none"> ◦ Varbinary(max) column ◦ BLOBs stored in NTFS file system ◦ Create a FILESTREAM filegroup • Advantages <ul style="list-style-type: none"> ◦ Better performance for BLOBs > 1 MB and no limit ◦ Fully integrated: management, security, referential and transactional integrity ◦ Full text searches • Disadvantages <ul style="list-style-type: none"> ◦ Applications cannot access BLOBs directly ◦ Must be stored on hard disk or SAN |
|---|

FileTables

FileTables was introduced in SQL Server 2012 as an extension to the FILESTREAM features that solve some of that feature's limitations.

Benefits and Implementation

When you use FILESTREAM columns, desktop applications such as Word cannot open BLOBS as they would normal files, because you can only access FILESTREAM BLOBS programmatically through Transact-SQL or Win 32 APIs. With FileTables, you can open and manipulate the contents of BLOBs from such applications.

Also, FILESTREAM BLOBS must be stored on a hard drive that is local to the database server. With FileTables, the storage location can be a shared folder on a file server that is remote to the database server.

A FileTable is a database table that has a specific schema. It includes a **varchar(max)** column with the FILESTREAM attribute enabled. It also includes a set of metadata columns that describe the BLOBS. These columns include the file size, the creation time, the last write time, and so on.

Advantages and Disadvantages

Because FileTables are an extension of the FILESTREAM feature, they realize many of the advantages associated with FILESTREAM. However, they also have extra advantages:

- Depending on your configuration, applications such as Word and Excel® can access files in the shared file system location. They can read, write, create and delete files, and their changes will be automatically propagated into the FileTable in the database.
- The file system location need not be on a hard drive installed on the database server. Instead, it could be a shared folder on a file server.

You should also consider the following disadvantage:

- Because a FileTable is a separate database table with a fixed schema, you cannot integrate the BLOBS as columns in another table. For example, product manuals must be stored as rows in a separate FileTable, and not as a column in the Products table. Instead you must use a foreign key relationship to associate a product with its manual. This may have implications for referential and transactional integrity.

- FileTable implementation
 - An extension to FILESTREAM
 - Varchar(max) column with FILESTREAM enabled
 - Can use a shared folder on a remote server
- FileTable benefits
 - Applications can read and manipulate BLOB content
- Advantages
 - All the advantages of FILESTREAM, plus applications such as Word and Excel can access files
 - Can read, write and delete
- Disadvantages
 - Separate database table with fixed schema

Demonstration: BLOBs in the Adventure Works Database

Demonstration Steps

Investigate BLOB Storage in the Adventure Works Database

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in **Server name**, type **MIA-SQL**, and then click **Connect**.
3. On the **File** menu, point to **Open**, and then click **Project/Solution**.
4. In the **Open Project** dialog box, navigate to **D:\Demofiles\Mod16\Demo**, click **demo.ssmsln**, and then click **Open**.
5. In Solution Explorer, double-click the **1 - AdventureWorks BLOBs.sql** script file.

6. Select the code under the **Step 1** comment, and then click **Execute**.
7. Select the code under the **Step 2** comment, and then click **Execute**.
8. Select the code under the **Step 3** comment, and then click **Execute**.
9. Select the code under the **Step 4** comment, and then click **Execute**. Note that no results are returned.
10. Close Microsoft SQL Server Management Studio, without saving any changes.

Check Your Knowledge

Question	
<p>You want to store résumés, which are Word documents, in the Human Resources database. You want to store each document in a column of the Employees table. A typical résumé is around 50 KB in size and you want to maximize performance, and ensure referential and transactional integrity. Which of the following approaches should you use?</p>	
Select the correct answer.	
	Store documents as BLOBs within the database.
	Store documents on a separate file server. In the database, create a varchar() column that links an employee record to the correct résumé.
	Use the FILESTREAM feature.
	Use a FileTable.

Lesson 2

Working with FILESTREAM

The FILESTREAM feature, together with FileTables, enable database administrators to combine the performance advantages of BLOBS stored on the file system, with the close integration of data when BLOBs are stored in the database. Now that you understand when to use FILESTREAM and FileTables, this lesson discusses their prerequisites and how to implement them in your database.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the requirements and limitations of FILESTREAM.
- Enable FILESTREAM for a SQL Server instance, a database, and an individual table.
- Describe the requirements and limitations of FileTable.
- Create FileTables in a database.
- Write queries that determine BLOB locations for FileTables and FILESTREAM columns.
- Configure and use FILESTREAM and FileTables.

Considerations for Using FILESTREAM

The advantages of the FILESTREAM technology make it an attractive option to store large BLOBS that are tightly integrated with database rows. Before implementing this technology, however, consider the following issues:

- For each table with FILESTREAM columns, you must also create a uniqueidentifier ROWGUID column.
- FILESTREAM requires at least one FILESTREAM filegroup. This special kind of filegroup contains file system directories instead of the BLOBS themselves. These directories are called data containers. Create a filegroup before you create a FILESTREAM column.
- FILESTREAM filegroups should be placed on a separate volume from the operating system, page files, the database, the transaction logs, and the **tempdb** for optimal performance.
- BLOBs in FILESTREAM columns are automatically included in database backups and restores, so you do not need a separate maintenance regime for FILESTREAM data.
- If you expect BLOBs in a column to be smaller than 1 MB, you might obtain better performance by using a **varbinary(max)** column without FILESTREAM. This is because small BLOBs can be stored in the same data page as the rest of the row.
- FILESTREAM is not compatible with database mirroring.
- If you are using transparent database encryption for a database, BLOBs stored in FILESTREAM columns are not encrypted.
- FILESTREAM is not supported by Always Encrypted.

- Each table that uses FILESTREAM columns must contain a uniqueidentifier ROWGUID column
- A FILESTREAM filegroup is required
- For best performance, the filegroup should be physically separated from other important files
- FILESTREAM BLOBS are backed up and restored with the database
- Small BLOBs may perform better without FILESTREAM enabled
- FILESTREAM is not compatible with database mirroring
- Transparent encryption does not protect FILESTREAM BLOBs
- FILESTREAM is not supported by Always Encrypted

Enabling FILESTREAM

FILESTREAM is not enabled by default in SQL Server. To use it, you must complete three configuration tasks:

1. Enable FILESTREAM for the SQL Server instance.
2. Set up a FILESTREAM filegroup in the database.
3. Create a FILESTREAM column in a table.

- Enable FILESTREAM for the instance
 - EXEC sp_configure filestream_access_level, 2 RECONFIGURE
 - Create a FILESTREAM filegroup for the database
 - Create a table with a FILESTREAM column

Enabling FILESTREAM for the SQL Server Instance

First, use SQL Server Configuration Manager to enable FILESTREAM:

1. Start SQL Server Configuration Manager.
2. In the list of services on the left, click **SQL Server Services**.
3. Locate the SQL Server instance you want to configure and double-click it.
4. In the **Properties** dialog for the SQL Server instance, click the **FILESTREAM** tab.
5. Select **Enable FILESTREAM for Transact-SQL access**.
6. If you want to use Win32 APIs to access BLOBS, select **Enable FILESTREAM for file I/O access**, and then click **OK**.

You must also configure the FILESTREAM access level by using the `sp_configure` stored procedure. There are three possible access levels:

- **0.** This value disables FILESTREAM for the instance.
- **1.** This value enables FILESTREAM access for Transact-SQL clients.
- **2.** This value enables FILESTREAM access for Transact-SQL and Win32 streaming clients.

Use the following code to configure the FILESTREAM access level:

Setting the Access Level by Using `sp_configure`

```
EXEC sp_configure filestream_access_level, 2
RECONFIGURE;
GO
```

After completing these configuration steps, restart the SQL Server instance.

Creating a FILESTREAM Filegroup

Your database must include at least one FILESTREAM filegroup. To create such a database, specify the `CONTAINS FILESTREAM` clause as in the following example:

This code example creates a new database with one filegroup that supports FILESTREAM:

Creating a Database with a FILESTREAM Filegroup

```
CREATE DATABASE HumanResources
ON
PRIMARY ( NAME = HR1,
    FILENAME = 'C:\databases\HRDB1.mdf'),
FILEGROUP FileStreamGroup1 CONTAINS FILESTREAM( NAME = HR3,
    FILENAME = 'D:\databases\filestream1')
LOG ON ( NAME = HRlog1,
    FILENAME = 'E:\TransLogs\HRlog1.ldf')
GO
```

Creating a FILESTREAM Column

Now that you have an instance and a database that support FILESTREAM, you can create tables with FILESTREAM columns. These columns must use the **varbinary(max)** data type, and you must specify the **FILESTREAM** attribute.

The following code creates a new table named "Employees" with a FILESTREAM column named "Resume":

Creating a FILESTREAM-Enabled Table

```
CREATE TABLE HumanResources.dbo.Employees
(
    [Id] [uniqueidentifier] ROWGUIDCOL NOT NULL UNIQUE,
    [EmployeeNumber] INTEGER UNIQUE,
    [FirstName] STRING NOT NULL,
    [LastName] STRING NOT NULL,
    [Resume] VARBINARY(MAX) FILESTREAM NULL
);
GO
```

Considerations for Using FileTables

FileTables are an extension to the FILESTREAM technology. Therefore, some of the prerequisites for FileTables are the same as for FILESTREAM—for example, a FILESTREAM filegroup is required.

The prerequisites for FileTables include:

- FILESTREAM must be enabled at the instance level. See the previous topic for how to configure the prerequisite.
- A FILESTREAM filegroup must be enabled at the database level. See the previous topic for how to configure the prerequisite.
- Nontransactional access must be configured at the database level. A key advantage of FileTables is that they enable applications to access BLOBs without going through the transactional system of SQL Server. You must configure this access and choose the access level. You can choose to enable full access, read-only access, or disable the access. See the next topic for how to enable nontransactional access.
- A directory for FileTables must be configured at the database level. When you create a FileTable, a folder is created as a child of the directory that will store BLOBs. See the next topic for how to configure this directory.

- FileTable prerequisites
 - FILESTREAM enabled
 - FILESTREAM filegroup configured
 - Nontransactional access configured
 - FileTable directory configured
- FileTable limitations
 - No table partitioning
 - No database replication
 - No transactional rollbacks and point-in-time recovery
 - Limited access to FileTables after AlwaysOn availability group failover
 - No DML INSTEAD OF triggers
 - No indexed views for FileTables

FileTables do not support all of the SQL Server features that other tables support, and some SQL Server features are partially supported.

- The following SQL Server features cannot be used with FileTables:
 - Table partitioning
 - Database replication
 - Transactional rollbacks and point-in-time recovery
- The following SQL Server features can be used with some limitations:
 - If a database includes a FileTable, failover works differently for AlwaysOn availability groups. If failover occurs, you have full access to the FileTable on the primary replica but no access on readable secondary replicas.
 - FileTables do not support INSTEAD OF triggers for Data Manipulations Language (DML) operations. AFTER triggers for DML operations are supported, and both AFTER and INSTEAD OF triggers are supported for Data Definition Language (DDL) operations.
 - You cannot include FileTables in indexed views.

Enabling FileTables

Before you can use FileTables, FILESTREAM must be enabled at the instance level, and a FILESTREAM filegroup created as described previously in this lesson. In addition, you must complete the tasks described in the following sections.

- Enabling Nontransactional Access
- Configuring a Directory for FileTables
- CREATE DATABASE HumanResources
WITH FILESTREAM (NON_TRANSACTED_ACCESS = FULL,
DIRECTORY_NAME = N'FileTableDirectory');

Enabling Nontransactional Access

Windows applications such as Word and Excel can obtain a file handle to BLOBs in the FileTable without supporting transactions. This means these applications can open BLOBs as they would open files in a typical Windows shared folder. This is called nontransactional access and is enabled at the database level.

Use the following query to check whether nontransactional access has been enabled for all the databases in an instance:

Checking for Nontransactional Access

```
SELECT DB_NAME(database_id), non_transacted_access, non_transacted_access_desc
  FROM sys.database_filestream_options;
GO
```

The available access levels for nontransactional access are DISABLED, READ_ONLY, and FULL. To enable FileTables and set the access level, use the FILESTREAM option when you create or alter a database.

In this example, full nontransactional access is enabled for a new database called HumanResources:

Enabling Nontransactional Access for a New Database

```
CREATE DATABASE HumanResources
  WITH FILESTREAM ( NON_TRANSACTED_ACCESS = FULL );
GO
```

Configuring a Directory for FileTables

You configure a directory for FileTables by using the DIRECTORY_NAME attribute in the FILESTREAM option when you create or alter a database.

In the following code, a FileTable directory is configured for a pre-existing database named HumanResources:

Configuring a Directory

```
ALTER DATABASE HumanResources
    SET FILESTREAM ( NON_TRANSACTED_ACCESS = FULL ,
    DIRECTORY_NAME = N'FileTableDirectory' );
GO
```

 **Note:** You can also enable and configure FileTable prerequisites by using SSMS. These options appear on the **Options** tab of the database **Properties** dialog box.

FILESTREAM Access

You can use Transact-SQL built-in system functions to determine the file system paths and IDs for folders that store FILESTREAM and FileTable BLOBS.

- Find the file system paths, and IDs FILESTREAM and FileTable folders
 - PathName()
 - File TableRootPath()
 - GetFileNameSpacePath()

The PathName Function

To access a BLOB stored in a FILESTREAM table, SELECT the column in a query. In an application, you must use further code to make use of the data, such as formatting an image BLOB for display on a webpage. Occasionally, you may want to determine the path to the BLOB on the file system; in this case, use the **PathName()** function.

Use the **@option** argument to control the format of the returned path. Possible values are:

- **0.** The path is returned in NetBIOS format. This is the default.
- **1.** The path is returned as stored without any conversion.
- **2.** The path is returned as a Universal Naming Convention (UNC) path.

The following query uses the **PathName()** function to locate a BLOB on the filesystem:

Determining BLOB Paths

```
SELECT FirstName, LastName, Photo.PathName()
FROM dbo.Employees
WHERE LastName = 'Smith'
ORDER BY LastName;
```

The FileTableRootPath Function

You can use this function to locate the root folder for a specific FileTable or for the current database. This is useful information, because FileTable allows direct access to BLOBS on the file system—you can pass this path to applications so they can edit files.

The following example returns the FileTable root path for the HumanResources database:

FileTableRootPath Function

```
USE HumanResources;
GO
SELECT FileTableRootPath();
```

The GetFileNamespacePath Function

This function returns the UNC path to a specific BLOB or directory in a FileTable. The **is_full_path** argument specifies whether the returned value should be a full or a relative path. The **@option** argument is used for the **PathName()** function.

The following query returns the relative paths to all BLOBS in a FileTable named Images:

GetFileNamespacePath Function

```
USE HumanResources;
GO
SELECT file_stream.GetFileNamespacePath() AS [Relative Path] FROM Images;
GO
```

Demonstration: Configuring FILESTREAM and FileTables

In this demonstration, you will see how to enable and create FILESTREAM columns and FileTables.

Demonstration Steps

Enable FILESTREAM at the Instance Level

1. On the Start screen, type **SQL Server 2016 Configuration Manager**, and then click **SQL Server 2016 Configuration Manager**.
2. In the **User Account Control** dialog box, click **Yes**.
3. In the left pane, click **SQL Server Services**.
4. In the right pane, right-click **SQL Server (MSSQLSERVER)**, and then click **Properties**.
5. In the **SQL Server (MSSQLSERVER) Properties** dialog box, on the **FILESTREAM** tab, check that the **Enable FILESTREAM for Transact-SQL access** check box is selected, and then click **OK**.
6. Close SQL Server Configuration Manager.

Configure FILESTREAM and FileTables in the AdventureWorks Database

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in **Server name**, type **MIA-SQL**, and then click **Connect**.
3. On the **File** menu, point to **Open**, and then click **Project/Solution**.
4. In the **Open Project** dialog box, navigate to **D:\Demofiles\Mod16\Demo**, click **demo.ssmssln**, and then click **Open**.
5. In Solution Explorer, double-click the **2 - Configuring FILESTREAM and FileTables.sql** script file.
6. Select the code under the **Step 1** comment, and then click **Execute**.
7. Select the code under the **Step 2** comment, and then click **Execute**.
8. Select the code under the **Step 3** comment, and then click **Execute**.
9. Select the code under the **Step 4** comment, and then click **Execute**.

10. Select the code under the **Step 5** comment, and then click **Execute**.
11. Select the code under the **Step 6** comment, and then click **Execute**.
12. Select the code under the **Step 7** comment, and then click **Execute**.
13. Select the code under the **Step 8** comment, and then click **Execute**.
14. Select the code under the **Step 9** comment, and then click **Execute**.

Keep Microsoft SQL Server Management Studio open for the next demonstration.

Sequencing Activity

You have no FILESTREAM or FileTable prerequisites configured. You want to create a FileTable for BLOB storage. Put the following steps in order by numbering each to indicate the correct order.

	Steps
	Enable FILESTREAM at the instance level.
	Create a FILESTREAM filegroup at the database level.
	Configure nontransactional access at the database level.
	Configure a directory for FileTables at the database level.
	Start creating FileTables.

Lesson 3

Using Full-Text Search

SQL Server has industry-leading indexing and querying performance optimized to handle structured, tabular data. If you use large **varchar()** columns to store long text fragments, or use **varbinary(max)** columns to store BLOBS, SELECT queries that use predicates such as LIKE might not perform well, or might not return the rows you intended. The Free-Text Search feature of SQL Server helps with these issues—it can analyze and index long text fragments in large **varchar()** columns and BLOBS in a way that is aware of language-specific linguistic rules, such as word boundaries and inflection.

In this lesson, you will see how to configure and query Full-Text Search and Semantic Search.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how Full-Text Search enables users to analyze text-based data in ways that are not possible with standard Transact-SQL queries.
- List the components of the Full-Text Search architecture and describe their role in index operations and queries.
- Configure Full-Text Search and create full-text indexes.
- Use predicates and functions to query a full-text index.
- Describe how Semantic Search enables users to analyze text-based data in ways that are not possible with Full-Text Search.

What is Full-Text Search?

Full-Text Search is an optional component of the SQL Server Database Engine that enables you to execute language-aware queries against character-based data. Each language, such as English, Japanese, or Thai, has different grammar and rules concerning the form of words and the boundaries between words. For example, in English, "running" and "ran" are both forms of the verb "run". It is difficult to create a LIKE predicate that would return rows that contain all forms of the verb "run". However, if you execute a full-text search for "run", items that contain "running" and "ran" are returned because full-text searches are aware of the rules of English grammar.

- Types of Full-Text Search
 - Simple term search
 - Prefix term search
 - Generation term search
 - Proximity term search
 - Thesaurus search
 - Weighted term search
- Performance
- Property-scoped Searches
- Language Support

Types of Full-Text Search

If you have a full-text index on a table, you can execute the following types of searches against the character-based data in the index:

- **Simple term search.** This type of search matches one or more specific words or phrases.
- **Prefix term search.** This type of search matches words that start with the character string you specify.
- **Generation term search.** This type of search matches inflectional forms of the words you specify.

- **Proximity term search.** This type of search matches an item when a specified word or phrase appears close to another specified word or phrase.
- **Thesaurus search.** This type of search matches words that are synonymous with the words you specify. For example, if you search for "run", a thesaurus search might match "jog".
- **Weighted term search.** This type of search matches the words or phrases you specify, and orders them so that some word matches appear higher in the list than others.

Full-text searches are not case-sensitive.

Performance

Full-text searches deliver much higher performance than LIKE predicates when executed against large blocks of text, such as **varchar(max)** columns. In addition, you cannot use LIKE predicates to search text in BLOBs, such as **varbinary(max)** columns. This restriction applies whether or not you are using FILESTREAM or FileTables.

Property-scoped Searches

Full-text search can index the properties of a file stored as a BLOB, in addition to its text. For example, Word supports an Author property for each Word document. You can use a full-text search to locate documents by a given author, even if you have not separately stored the author in a column, in a SQL Server table.

Language Support

Full-text search supports around 50 languages and distinguishes between dialects of the same language, such as American English and British English. For each language, the following components are used to analyze and index text:

- **Word breakers and stemmers.** A word breaker separates text into individual words by located word boundaries, such as spaces and periods. A stemmer conjugates verbs to ensure that different forms of the same word match.
- **Stoplists.** A stop word or noise word is one that does not help the search. A stoplist is a list of stop words for a given language. For example, in English, no one would search for the word "the" so it is removed from the index.
- **Thesaurus files.** Thesaurus files list synonyms to ensure that thesaurus searches match words that mean the same thing as the search term.
- **Filters.** Filters are components that understand the structure of a particular file type, such as a Word document or an Excel spreadsheet. Filters enable property-scoped searches by enabling SQL Server to index the properties of those file types.

You can use the sys.fulltext_languages catalog view to determine which languages are supported by full-text searches on your database server. For full details of this catalog view, see Microsoft Docs:

 **sys.fulltext_languages (Transact-SQL)**

<http://aka.ms/Hvzcir>

Components and Architecture of Full-Text Search

Full-Text Search Components

Multiple components work together to implement Full-Text Search. Some of these components are involved in creating the index, and some in executing queries.

Most of the components of SQL Server Full-Text Search run as part of the SQL Server process (sqlserver.exe). However, for security reasons, protocol handlers, filters, and word breakers are isolated in a separate process called the filter daemon host (fdhost.exe).

The components that run in the SQL Server process include:

- **Full-Text Gatherer.** This component runs the crawl threads that process content.
- **Full-Text Engine.** This component is part of the SQL Server query processor. It receives full-text queries and communicates with the index to locate matching items.
- **Thesaurus.** This component stores the language-dependent lists of synonyms that enable thesaurus search.
- **Stoplist.** This component defines stop words and removes them from queries and full-text indexes.
- **Indexer.** This component compiles the full-text index in a format that optimizes query delivery.

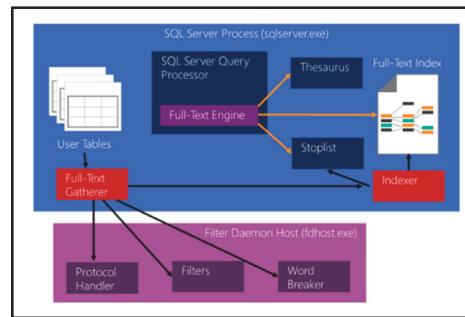
The components that run in the filter daemon host include:

- **Protocol Handlers.** These components communicate with the location where content is stored. Often, this location is a table in the database, but it can also be a file share and other types of location.
- **Filters.** These components analyze file structures, and locate file properties and body text.
- **Word breakers.** These components look for word boundaries in a specific language, such as spaces, commas, and periods.

Indexing Process

A full-text population operation is also called a crawl and can be initiated by a change to one of the indexed columns or on a schedule. When a crawl is initiated, these are the steps that are followed:

1. The full-text engine notifies the filter daemon host that a crawl is underway.
2. The full-text gatherer initiates crawl threads, each of which begins to crawl content and pass it to different components for processing.
3. The full-text engine reads a large quantity of data into memory in the form of user tables. These tables are the raw content of the character-based data in the columns of the index. Depending on the storage location, different protocol handlers are used to obtain this text.
4. Crawl threads pass BLOBS to filters. These analyze the content of the file and return text from the body and metadata fields.
5. Crawl threads pass text to word breakers that split long strings into words.
6. Individual word lists are passed to the indexer.
7. The indexer calls the stoplist to remove noise words from the index.



8. The indexer creates an inverted list of words and their locations in the columns, and stores this list in the full-text index.

Query Process

When a user executes a full-text query, the SQL Server Query Processor passes the request to the Full-Text Engine. The Full-Text Engine takes different steps to compile the query, depending on the type of search that was requested. For example:

- If the query is a generation term search, the Full-Text Engine performs stemming to identify alternate forms of the search terms.
- If the query is a thesaurus search, the Full-Text Engine calls the thesaurus to identify synonyms.
- If the query includes phrases, the Full-Text Engine calls word breakers.
- The Full-Text Engine removes noise words by calling the stoplist.
- The Full-Text Engine represents the query in the form of SQL operations—primarily as Streaming Table-Valued Functions (STVFs).

Once query compilation has been completed, the SQL operations are executed against the full-text index to retrieve results, which are returned to the client.

For more information about full-text search architecture, see Microsoft TechNet:

Full-Text Search Architecture

<https://aka.ms/Ulbrhm>

Configuring Full-Text Search

To implement Full-Text Search with optimal performance, consider the following issues to configure the full-text components:

- **Supported data types.** Full-text indexes can only include columns with the following data types: **char**, **varchar**, **nchar**, **nvarchar**, **text**, **ntext**, **image**, **xml**, **varbinary**, and **varbinary(max)**, including **varbinary(max)**, with FILESTREAM enabled.
- **Unique column.** A full-text index requires a unique, non-null column as a key index. For the best performance, this column should be of the **integer** data type.
- **Table support.** You can only create one full-text index for each database table, but the index can include multiple columns from that table.
- **Language support.** A single full-text index can include text in multiple languages. You specify a single language for each column in the index.
- **Filegroup placement.** Full-text crawls are disk-intensive operations, so you should consider creating a dedicated filegroup for full-text indexes. For maximized performance, separate this filegroup onto its own physical disk.
- **Managing updates.** By default, the Full-Text Engine is configured to update the index continuously as changes are made to the underlying column data. This ensures that the index is always up to date.

Considerations for Full-Text Search

- Data types: **char**, **varchar**, **nchar**, **nvarchar**, **text**, **ntext**, **image**, **xml**, **varbinary**, **varbinary(max)**, incl FILESTREAM
- Requires a unique, non-null column, ideally integer
- One full-text index per table
- Specify language for each column in the index
- Consider dedicated filegroup for full-text indexes
- Consider off-peak crawls to update the index
- Create a full-text catalog
- Create a full-text index

However, you may wish to schedule crawls to take place during off-peak hours or to manually initiate crawls. Schedules use the SQL Server Agent service to initiate crawls. Remember that the index may fall out of synchronization with the column data if a crawl has not taken place recently.

Creating a Full-Text Catalog

The first step you must take to configure Full-Text Search is to create a full-text catalog. This is a logical grouping for full-text indexes in the database.

The following code creates a full-text catalog in the HumanResources database:

Creating a Full-Text Catalog

```
USE HumanResources;
GO
CREATE FULLTEXT CATALOG HRCatalog;
GO
```

Creating a Full-Text Index

To create a full-text index, use the CREATE FULLTEXT INDEX statement. You must supply the following information:

- A name for the index.
- The name of the table on which to create the index.
- The names of the columns to include in the index.
- The name of the column that will act as the key index column.
- Optionally, a language for each column in the index.

The following code creates a full-text index in the HRCatalog. It uses the EmployeeID column as the key index and includes the FullName, EmailAddress, Skills, and Resume columns. The document type for Resume files is specified in the file_type column:

Creating a Full-Text Index

```
CREATE FULLTEXT INDEX ON HumanResources.Employees
(
    FullName
        Language 1033,
    EmailAddress
        Language 1033,
    Skills
        Language 1033,
    Resume TYPE COLUMN [file_type]
        Language 1033
)
KEY INDEX EmployeeID
    ON HRCatalog;
GO
```

Querying a Full-Text Index

To execute a query that makes use of a full-text index, you must use the CONTAINS or FREETEXT predicates, often in a WHERE clause. Alternatively, you can use the CONTAINSTABLE or FREETEXTTABLE functions, which return tables of rows that match your search and can be used in FROM clauses.

CONTAINS and FREETEXT Predicates

The CONTAINS and FREETEXT predicates return TRUE or FALSE. You can use them in WHERE clauses or HAVING clauses of a SELECT statement to ensure that only matching rows are included in the result set. Other predicates, such as LIKE or BETWEEN, can be used in the same clause if required. When you use CONTAINS or FREETEXT you can specify a column in the index that must match a list of columns, or that all columns in the index should be searched.

Use CONTAINS to locate precise or fuzzy matches to words and phrases. You can also use CONTAINS to perform proximity term searches or weighted term searches.

In the following code, the query returns all Employees in the Sales department that have the phrase "Team Management" in the Skills column. This is an example of a simple term search:

Simple Term Search

```
SELECT EmployeeID, FirstName, LastName
FROM HumanResources.Employees
WHERE Department = 'Sales'
AND CONTAINS(Skills, 'Team Management');
```

In the following example, the query returns all Employees with forms of the verb "analyze" in their résumé. Results would include employees with the words "analyzing" and "analyzed" in their résumés. The Resume column may be a BLOB column that uses the **varbinary(max)** column, either with FILESTREAM enabled or with BLOBs stored in the database.

Generation Term Search

```
SELECT EmployeeID, FirstName, LastName
FROM HumanResources.Employees
WHERE CONTAINS(Resume, ' FORMSOF (INFLECTIONAL, analyze) '');
```

Use FREETEXT to match the meaning, rather than the exact wording of single words, phrases, or sentences. FREETEXT searches use the thesaurus to match meaning.

The following example uses the FREETEXT predicate to perform a thesaurus search:

Thesaurus Search

```
SELECT EmployeeID, FirstName, LastName
FROM HumanResources.Employees
WHERE FREETEXT (Resume, 'Project Management' );
```

CONTAINSTABLE and FREETEXTTABLE Functions

These functions return tables of rows that match the full-text query you specify. You can reference these tables in the FROM clause of a SELECT statement. The tables returned always include the following columns:

- CONTAINS and FREETEXT
 - Returns TRUE or FALSE
 - Use them to find matches
 - Specify one or more columns
- CONTAINSTABLE and FREETEXTTABLE
 - Returns a table
 - Tables can then be referenced from a SELECT statement
 - Contains a KEY and RANK column
- Using a NEAR Proximity Term
 - Search for rows that contain two words or phrases close to each other

- **KEY.** This column returns the unique value of the key index column of the full-text index.
- **RANK.** This column contains a rank value that describes how well the row matched the query. The higher the ranks value, the better the match.

In the following example, a weighted term search is performed by using the CONTAINSTABLE function. The results are joined to the original table to return the rank value for each product:

Using CONTAINSTABLE for a Weighted Term Search

```
SELECT FT_TBL.Name, KEY_TBL.RANK
  FROM Production.Product AS FT_TBL
    INNER JOIN CONTAINSTABLE(Production.Product, Name,
      'ISABOUT (frame WEIGHT (.8),
      wheel WEIGHT (.4), tire WEIGHT (.2) )') AS KEY_TBL
        ON FT_TBL.ProductID = KEY_TBL.[KEY]
 ORDER BY KEY_TBL.RANK DESC;
```

In the following example, the FREETEXTTABLE function is called to perform a thesaurus search. The results are joined with the original table to display the rank value with the search column:

Using the FREETABLE Function for a Thesaurus Search

```
SELECT KEY_TBL.RANK, FT_TBL.Description
  FROM Production.ProductDescription AS FT_TBL
    INNER JOIN
      FREETEXTTABLE(Production.ProductDescription, Description,
        'perfect all-around bike') AS KEY_TBL
          ON FT_TBL.ProductDescriptionID = KEY_TBL.[KEY]
 ORDER BY KEY_TBL.RANK DESC;
```

For more information about CONTAINS, FREETEXT, CONTAINSTABLE, and FREETEXTTABLE see Microsoft Docs:

Query with Full-Text Search

<http://aka.ms/Ai7pzu>

Using a NEAR Proximity Term

When you execute queries that include CONTAINS or CONTAINSTABLE, you can search for rows that contain two words or phrases close to each other in the full-text index. To do this, you use the NEAR keyword, also known as the custom proximity term.

When you use the custom proximity term, you can specify the maximum number of nonsearch terms that separate your search terms. This is known as the maximum distance between search terms. You can also define whether the returned result must contain the search terms in the specified order.

In the following example, the query will return employees if the words "Project" and "Management" appear with five or fewer terms separating them in the Resume column:

Using the Custom Proximity Predicate

```
SELECT EmployeeID, FirstName, LastName
  FROM HumanResources.Employees
 WHERE CONTAINS(Resume, 'NEAR((Project, Management), 5)');
```

For more information on the custom proximity term, see Microsoft Docs:

Search for Words Close to Another Word with NEAR

<http://aka.ms/Abw20y>

Demonstration: Configuring and Using Full-Text Search

In this demonstration, you will see how to create a full-text index and use it for generation term searches.

Demonstration Steps

Create and Use a Full-Text Index

1. In Solution Explorer, double-click the **3 - Configuring and Using Full-Text Search.sql** script file.
2. Select the code under the **Step 1** comment, and then click **Execute**.
3. Select the code under the **Step 2** comment, and then click **Execute**.
4. Select the code under the **Step 3** comment, and then click **Execute**.
5. Select the code under the **Step 4** comment, and then click **Execute**.
6. Select the code under the **Step 5** comment, and then click **Execute**.
7. Select the code under the **Step 6** comment, and then click **Execute**.
8. Close Microsoft SQL Server Management Studio, without saving any changes.

Enabling and Using Semantic Search

Semantic Search provides deep insight into character-based data, including the data stored in large BLOBs, by extracting and indexing statistically relevant key phrases. You can use Semantic Search to identify documents that are similar or related, based on the meaning of their content. It is this emphasis on meaning, rather than individual search terms, which separates Semantic Search from Full-Text Search.

- Semantic search extends full-text search
 - Identify similar or related documents
 - Enabling semantic search functionality
 - Installing the Semantic Language Statistic database
 - Adding semantic search to a full-text index
 - Using semantic search functions in queries
 - SEMANTICKEYPHRASETABLE
 - SEMANTICSIMILARITYTABLE
 - SEMANTICSIMILARITYDETAILSTABLE

What is Semantic Search?

Semantic Search extends the capabilities of Full-Text Search so you can identify documents that are similar or related in some way. For example, you could use Semantic Search to identify the résumés in a FileTable that relate to a specific job role. Although a standard full-text query will reveal résumés that contain similar keywords or phrases, these searches may not find relevant résumés where the author has not used the specified keywords that are contained in the search term. By identifying deeper patterns of meaning, Semantic Search can provide a results set that more accurately matches the search query.

Semantic Search uses a database named the Semantic Language Statistics database, which contains the statistical models that are used to perform semantic searches.

 **Note:** Semantic Search does not support as many languages as a full-text index. To view the list of supported languages for Semantic Search, query the sys.fulltext_semantic_languages catalog view.

Enabling Semantic Search Functionality

Semantic Search is an extension to Full-Text Search and uses full-text indexes. In addition, you must install the Semantic Language Statistics database, which contains the information SQL Server uses to analyze meaning in text. You must also modify a full-text index to support Semantic Search.

For more information on how to install the Semantic Language Statistics database, see Microsoft Docs:

Install and Configure Semantic Search

<http://aka.ms/Wuhba4>

After the Semantic Language Statistics database is configured, you can use the **CREATE FULLTEXT INDEX** statement or the **ALTER FULLTEXT INDEX** statement to create a full-text index that includes Semantic Search.

The following code example adds Semantic Search to an existing full-text index on the Document table in the AdventureWorks database:

Using ALTER FULLTEXT INDEX to Add Semantic Search

```
ALTER FULLTEXT INDEX ON Production.Document
    ALTER COLUMN Document
        ADD Statistical_Semantics;
GO
```

Using Semantic Search Functions in Queries

Once you have configured Semantic Search, you can use three functions to use it in queries. These functions identify key phrases in a document, in addition to documents that are similar to each other, because they share these key phrases. The functions are as follows:

- **SEMANTICKEYPHRASETABLE**. This function returns a table of key phrases from the column or columns that you specify. These columns can include BLOBs in **varchar(max)** columns, including FILESTREAM columns and document columns in FileTables. The returned table includes the following columns:
 - **Document_key**. This is the key index value for the returned document in the underlying full-text index.
 - **Keyphrase**. This is the phrase that the search has identified as key to the meaning of the document.
 - **Score**. This is a weighting value that indicates the importance of the phrase in the document. The value is between 0 and 1.
- **SEMANTICSIMILARITYTABLE**. This function returns a table of documents that are semantically related to a specified document in the same full-text indexed table. You specify the table to search, the column or columns to search, and the key index of the document to compare. The returned table includes the following columns:
 - **Matched_document_key**. This is the key index value for the matched document.
 - **Score**. This is a weighting value that indicates the closeness of the match. The value is between 0 and 1.
- **SEMANTICSIMILARITYDETAILTABLE**. This function returns the key phrases that make two documents similar. Having used **SemanticSimilarityTable** to find similar documents, you can use this function to determine the phrases that the similar documents share. The returned table includes the following columns:
 - **Keyphrase**. This is the phrase that is shared between the two documents you have specified.
 - **Score**. This is a weighting value that indicates how important this key phrase is in its similarity between the two documents.

The following example uses the **SEMANTICKEYPHRASETABLE** function to return the top 10 key phrases from a specific document in the Resume column of the Employees table. The document is specified by using the @EmployeeId parameter, which is the key index of a row in the Employees table.

SEMANTICKEYPHRASETABLE

```
SELECT TOP(10) KeyPhraseTable.keyphrase
FROM SEMANTICKEYPHRASETABLE
(
    HumanResources.Employees,
    Resume,
    @EmployeeId
) AS KeyPhraseTable
ORDER BY KEYP_TBL.score DESC;
```

Check Your Knowledge

Question
You have a full-text index set up on the HumanResources.Employees table that includes the Resume column. You want to locate employees who have management skills. You want to search the Resume column for the word "manage" and you want résumés with the words "manager", "managed", and "managing" to be included in the results. What kind of search should you use?
Select the correct answer.
<input type="radio"/> A simple term search.
<input type="radio"/> A generation term search.
<input type="radio"/> A proximity term search.
<input type="radio"/> A thesaurus search.
<input type="radio"/> A weighted term search.

Lab: Storing and Querying BLOBs and Text Documents in SQL Server

Scenario

Your manager has asked you to evaluate and optimize the performance of queries against the **LargePhoto** column in the **Production.ProductPhotos** table. You have decided that, because BLOBs in this column are often larger than 1 MB, it will be advantageous to create a FILESTREAM column and move the existing data into the new column.

You have also been asked to create a FileTable, with a corresponding shared folder, so users can store documents by using Word and other desktop applications. These files will be accessible through the file share and database queries.

Finally, you have also been asked to create a full-text index on the **Description** column in the **Production.ProductDescriptions** table so that generation term queries and thesaurus queries can be used.

Objectives

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

- Enable FILESTREAM and move BLOB data into a FILESTREAM column.
- Enable FileTables and create a FileTable.
- Create and query a full-text index.

Estimated Time: 45 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Enabling and Using FILESTREAM Columns

Scenario

Having decided to move BLOB data into a FILESTREAM column, you will now implement that strategy.

The main tasks for this exercise are as follows:

1. Prepare the Environment
2. Enable FILESTREAM for the SQL Server Instance
3. Enable FILESTREAM for the Database
4. Create a FILESTREAM Column
5. Move Data into the FILESTREAM Column

► Task 1: Prepare the Environment

Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab16\Starter** folder as Administrator.

► **Task 2: Enable FILESTREAM for the SQL Server Instance**

1. Using SQL Server Configuration Manager, enable **FILESTREAM** for the SQL Server (MSSQLSERVER) instance. Enable **FILESTREAM** for file I/O access and give all remote clients access to FILESTREAM data.
2. Restart the SQL Server (MSSQLSERVER) service.
3. Open the project file **D:\Labfiles\Lab16\Starter\Project\Project.ssmssln** and the T-SQL script **Lab Exercise 1.sql**. Ensure that you are connected to the **AdventureWorks** database.
4. Write and execute a script that uses the `sp_configure` stored procedure to set the **FILESTREAM** access level to 2.

► **Task 3: Enable FILESTREAM for the Database**

1. In SQL Server Management Studio, write a query to add a FILESTREAM filegroup called **AdWorksFilestreamGroup** to the **AdventureWorks2016** database.
2. Write a query to add a file named **D:\FilestreamData** to the **AdventureWorks** database.
- **Task 4: Create a FILESTREAM Column**
1. In SQL Server Management Studio, write a query to add a new UNIQUEIDENTIFIER, non-nullable row GUID column called **PhotoGuid** to the **Production.ProductPhoto** table.
2. Write a query to enable **FILESTREAM** for the **Product.ProductPhoto** table, and add BLOBs to the **AdworksFilestreamGroup**.
3. Write a query to add a new column called **NewLargePhoto** to the **Production.ProductPhoto** table. Ensure the new column has FILESTREAM enabled and is a nullable varbinary(max) column.

► **Task 5: Move Data into the FILESTREAM Column**

1. In SQL Server Management Studio, write a query to copy all data from the **LargePhoto** column into the **NewLargePhoto** column.
2. Write a query to drop the **LargePhoto** column from the **Production.ProductPhoto** table.
3. Write a query that used the `sp_rename` stored procedure to change the name of the **NewLargePhoto** column **to LargePhoto**.
4. Close the Lab Exercise 1.sql script.

Results: At the end of this exercise, you will have:

Enabled FILESTREAM on the SQL Server instance.

Enabled FILESTREAM on a database.

Moved data into the FILESTREAM column.

Exercise 2: Enabling and Using FileTables

Scenario

You have decided to allow users to be able to store files in the database, and will now implement that strategy with FileTables.

The main tasks for this exercise are as follows:

1. Enable Nontransactional Access
2. Create a FileTable
3. Add a File to the FileTable

► Task 1: Enable Nontransactional Access

1. In SQL Server Management Studio, open the T-SQL script **Lab Exercise 2.sql**.
2. Write a query that uses the `sys.database_filestream_options` system view to display whether nontransacted access is enabled for each database in the instance.
3. Write a query that enables nontransacted access for the **AdventureWorks2016** database. Set the transacted access level to full and the directory name to “**FileTablesDirectory**”.

► Task 2: Create a FileTable

1. In SQL Server Management Studio, write a query to create a new FileTable in the **AdventureWorks2016 database**. Name the FileTable “**DocumentStore**” and use a FileTable directory named “**DocumentStore**”.
2. Write a query that uses the `sys.database_filestream_options` system view to display whether nontransacted access is enabled for each database in the instance.
3. Write a query that uses the `sys.filetables` system view to list the FileTables in the **AdventureWorks** database.

► Task 3: Add a File to the FileTable

1. In SQL Server Management Studio, write a query that uses the `FileTableRootPath()` function to find the path to the file share for the **DocumentStore** filetable.
2. Copy and paste the path you determined into the address bar of a new File Explorer window.
3. Create a new text document called **DocumentStoreTest** in the file table shared folder.
4. In SQL Management Studio, write a query that displays all rows in the DocumentStore FileTable.

Results: At the end of this exercise, you will have:

Enabled nontransactional access.

Created a FileTable.

Added a file to the FileTable.

Exercise 3: Using a Full-Text Index

Scenario

You will now create a full-text index on the **Description** column in the **Production.ProductDescriptions** table so that generation term queries and thesaurus queries can be used.

The main tasks for this exercise are as follows:

1. Create a Full-Text Index
2. Using a Full-Text Index

► Task 1: Create a Full-Text Index

1. In SQL Server Management Studio, open the T-SQL script **Lab Exercise 3.sql**.
2. Execute the first SELECT query in the Lab Exercise 2.sql script file, which lists the tables that have a full-text index in the **Adventure Works2016** database.
3. Write a query that creates a new full-text catalog in the **Adventure Works2016** database with the name **ProductFullTextCatalog**.
4. Write a query that creates a new unique index called **ui_ProductDescriptionID** and indexes the **ProductDescriptionID** column in the **Production.ProductDescription** table.
5. Write a query that creates a new full-text index on the **Description** column of the **Production.ProductDescription** table. Use the **ui_ProductDescription** unique index and the **ProductFullTextCatalog**.

► Task 2: Using a Full-Text Index

1. In SQL Server Management Studio, write a script that executes a simple term query against the **Description** column in the **Production.ProductDescription** table. Locate rows that contain the word "Bike". Make a note of the number of rows returned.
2. Write a script that executes a generation term query against the **Description** column in the **Production.ProductDescription** table. Locate rows that contain the word "Bike". Make a note of the number of rows returned.
3. Write a script that returns rows from the previous generation term query but not terms from the previous simple terms query. Examine the Description text for these results.
4. Close Microsoft SQL Server Management Studio, without saving any changes.

Results: At the end of this exercise, you will have created a full-text index.

Question: How did the results of the simple term query you executed in Exercise 3, Task 2 differ from the results of the generation terms query?

Question: What did you notice about the results of the third query you ran against the full-text index?

Module Review and Takeaways

In this module, you have seen the different approaches database administrators can take to the storage of BLOBS, such as images, documents, and videos. You have learned how to enable and use the advance FILESTREAM technology and FileTables for BLOBs. You have also learned how to configure and use File-Text Search.

Module 17

SQL Server Concurrency

Contents:

Module Overview	17-1
Lesson 1: Concurrency and Transactions	17-2
Lesson 2: Locking Internals	17-14
Lab: Concurrency and Transactions	17-27
Module Review and Takeaways	17-31

Module Overview

Concurrency control is a critical feature of multiuser database systems; it allows data to remain consistent when many users are modifying data at the same time. This module covers the implementation of concurrency in Microsoft SQL Server®. You will learn about how SQL Server implements concurrency controls, and the different ways you can configure and work with concurrency settings.



Note: Transactions and locking are closely interrelated; it is difficult to explain either topic without reference to the other. This module covers transactions and concurrency before covering locking, but you will find some references to locking in the description of transactions.

Objectives

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

- Describe concurrency and transactions in SQL Server.
- Describe SQL Server locking.

Lesson 1

Concurrency and Transactions

This lesson focuses on how SQL Server implements concurrency and transactions. You will learn about different concurrency models, and the strengths and weaknesses of each model. You will then learn about different types of isolation levels and transaction internals.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe different models of concurrency.
- Identify concurrency problems.
- Implement isolation levels.
- Work with row versioning isolation levels.
- Describe how SQL Server implements transactions.
- Explain best practices when working with transactions.

Concurrency Models

Concurrency can be defined as a system's ability to allow multiple users to access or change shared data simultaneously. The greater the number of active users able to work on shared data, the greater the level of concurrency. As the level of concurrency increases, the likelihood of conflicting data operations (where two or more users attempt to access or amend the same data at the same time) also increases.

There are two different approaches to resolving data conflicts during concurrent operation; these are pessimistic and optimistic concurrency.

- Pessimistic concurrency:
 - Data integrity maintained using locks
 - Only one user can access a data item at once
 - Writers block readers and other writers; readers block writers
 - Optimistic concurrency:
 - Data is checked for changes before update
 - Minimal locking

Pessimistic Concurrency

The pessimistic concurrency model assumes that conflicting data operations will occur frequently. In this model, locks are used to ensure that only one user can access one data item at a time. While a data item is locked to one user, other users cannot access it. A pessimistic concurrency model exhibits the following properties:

- Data being read is locked, so that no other user can modify the data.
- Data being modified is locked, so that no other user can read or modify the data.
- The number of locks acquired is high because every data access operation (read/write) acquires a lock.
- Writers block readers and other writers. Readers block writers.

The pessimistic concurrency model is suitable for a system where:

- Data contention is high.
- Locks are held for a short period of time.

- The cost of preventing conflicts with locks is lower than the cost of rolling back the change, in the case of a concurrency conflict.

Optimistic Concurrency

The optimistic concurrency model assumes that conflicting data operations will occur infrequently. In this model, locks are not used; instead, the state of the affected data is recorded at the start of a data operation. This state is checked again at the end of the operation, before any changes are written. If the state has not changed, new changes are written. If the state has changed, the new changes are discarded and the operation fails. An optimistic concurrency model exhibits the following properties:

- Data being read is not locked; other users may read or modify the data.
- Data being modified is not locked; other users may read or modify the data.
- Before modified data is written, it is checked to confirm that it has not changed since being read; only if it has not changed will the changes be written.
- The number of locks acquired is low.

The optimistic concurrency model is suitable for a system where:

- Data contention is low.
- Data modifications may take long periods of time.
- The cost of rolling back and then retrying a change is lower than the cost of holding locks.
- Readers should not block writers.

SQL Server supports implementations of both optimistic concurrency and pessimistic concurrency. Pessimistic concurrency is the default concurrency model for the database engine. The In-Memory OLTP Engine uses a type of optimistic concurrency called row versioning; it does not implement pessimistic concurrency.

Concurrency Problems

There are several categories of problems that may occur when concurrency control is lacking or insufficient, and multiple sessions attempt to access or change the same data item.

Dirty Read

A dirty read occurs when one transaction reads a row that is in the process of being modified by another transaction. The reading transaction reads uncommitted data that may be changed later by a transaction modifying the data.

For example, user A changes a value from x to y, but does not finalize the change by committing the transaction. A second user, user B, reads the updated value y and performs processing based on this value. User A later changes the value again, from y to z, and commits the transaction. User B reads the uncommitted (dirty) value.

- Dirty read
 - Uncommitted data is included in results
- Lost update
 - Two concurrent updates; the first update is lost
- Non-repeatable read
 - Data changes between two identical SELECT statements within a transaction
- Phantom read
 - Data is read, then deleted before reading completes
- Double read
 - Data in a range is read twice because the range key value changes

Lost Update

A lost update occurs when one or more transactions simultaneously updates the same row, based on the same initial value. When this happens, the last transaction to update the row overwrites the changes made by other transaction(s), resulting in lost data.

For example, user C and user D select value x to update. User C first updates the value from x to y , and then user D updates the value x to z . The modifications made by user A are overwritten by user B, resulting in data loss.

Non-repeatable Read

A non-repeatable read occurs when a transaction reads different values for the same row each time the row is accessed. This happens when data is changed by another transaction in between two SELECT statements.

For example, user E begins a transaction that contains two similar SELECT statements, $s1$ and $s2$. The SELECT statement $s1$ reads value x , and then does some processing. Another user, user F, modifies value x to y while user E is executing other queries. When user E subsequently executes $s2$, the value y is returned instead of the initial x .

Phantom Read

A phantom read is a variation of a non-repeatable read. Phantom reads occur when one transaction carries out a DELETE operation or an INSERT operation against a row that belongs to the range of rows being read by another transaction.

For example, user G has two similar SELECT statements, $s1$ and $s2$, within a transaction; the SELECT statement $s1$ reads the count of rows as n , and then does other processing. Meanwhile, another user, user H, deletes a row from the range of rows being read by select statement $s1$ and $s2$. When user G returns to execute $s2$, the row count is $n-1$. The SELECT statement $s1$ returns a phantom read for a row that does not exist at the end of user G's transaction.

Double Read

A double read occurs when a transaction reads the same row value twice when reading a range of rows. This happens when the row value that defines the range is updated by another transaction while the range is being read.

For example, user I executes a SELECT statement that returns rows with values in a range a to z , that is implemented as an index scan. After the scan has read rows with value a , but before the scan completes, another user, user J, updates a row with value a to value z . The updated row is read again when the scan reaches rows with value z .

 **Note:** It is also possible for this issue to miss a row, but this is still referred to as a double read problem. In the example, a row could be updated from value z to value a while the scan was running.

Transaction Isolation Levels

You can use transaction isolation levels to control the extent to which one transaction is isolated from another, and to switch between pessimistic and optimistic concurrency models. Transaction isolation levels can be defined in terms of which of the concurrency problems are permitted.

A transaction isolation level controls:

- Whether locks should be acquired when data is being read and the type of locks to be acquired.
- The duration that the locks are held.
- Whether a read operation accessing rows being modified by another transaction:
 - Is blocked until the exclusive lock on the row is released.
 - Fetches the committed data present at the time the transaction started.
 - Reads the uncommitted data modification.

- Pessimistic isolation levels:
 - READ UNCOMMITTED
 - READ COMMITTED
 - READ_COMMITTED_SNAPSHOT OFF
 - REPEATABLE READ
 - SERIALIZABLE
- Optimistic (row versioning) isolation levels:
 - READ COMMITTED
 - READ_COMMITTED_SNAPSHOT ON
 - SNAPSHOT

The transaction isolation level controls only whether locks are to be acquired or not for read operations; write operations will always acquire an exclusive lock on the data they modify and hold the lock until the transaction finishes, whatever the isolation level of transaction.

Isolation levels represent a trade-off between concurrency and consistency of data reads. At lower isolation levels, more concurrent data access is possible, but you experience more concurrency problems. At higher isolation levels, concurrency is reduced, but you experience fewer concurrency problems.

Five isolation levels are available in SQL Server:

READ UNCOMMITTED

READ UNCOMMITTED is the lowest level of isolation available in SQL Server. The READ UNCOMMITTED isolation level has the following properties:

- No locks are taken for data being read.
- During read operations, a lock is taken to protect the underlying database schema from being modified.
- Readers do not block writers, and writers do not block readers; however, writers do block writers.
- All of the concurrency problems (dirty reads, non-repeatable reads, double reads, and phantom reads) can occur.
- Data consistency is not guaranteed.
- Not supported on FILESTREAM enabled databases.

READ COMMITTED

READ COMMITTED is the SQL Server default isolation level. The READ COMMITTED isolation level has the following properties when the READ_COMMITTED_SNAPSHOT database option is OFF (the default for SQL Server installations):

- Read locks are acquired and held until the end of the statement.
- Dirty reads are eliminated by allowing access to committed data only.

- Because read locks are held until the end of the statement, data can be changed by other transactions between individual statements within the current transaction, resulting in non-repeatable reads, double reads, and phantom reads.

When the READ_COMMITTED_SNAPSHOT database option is ON (the default for Azure SQL Database), the READ COMMITTED isolation level has the following properties:

- Row versioning is used to provide statement-level read consistency. Because each statement in a transaction executes, a snapshot of old data is taken and stored in version store. The snapshot is consistent until the statement finishes execution.
- Read locks are not held because the data is read from the version store, and not from the underlying object.
- Dirty reads do not occur because a transaction reads only committed data, but non-repeatable reads and phantom reads can occur during a transaction.

READ COMMITTED isolation is supported on FILESTREAM enabled databases.

REPEATABLE READ

REPEATABLE READ has the following properties:

- Read locks are acquired and held until the end of the transaction. Therefore, a transaction cannot read uncommitted data and cannot modify the data being read by other transactions until that transaction completes.
- Eliminates non-repeatable reads. Phantom reads and double reads still occur. Other transactions can insert or delete rows in the range of data being read.
- Not supported on FILESTREAM enabled databases.

SERIALIZABLE

SERIALIZABLE is the highest level of isolation available in SQL Server. It has the following properties:

- Range locks are acquired on the range of values being read and are held until the end of the transaction.
- Transactions cannot read uncommitted data, and cannot modify the data being read by other transactions until the transaction completes; another transaction cannot insert or delete the rows in the range of rows being read.
- Provides lowest level of concurrency.
- Not supported on FILESTREAM enabled databases.

SNAPSHOT

SNAPSHOT isolation is based on an optimistic concurrency model. SNAPSHOT isolation has the following properties:

- Uses row versioning to provide transaction-level read consistency. A data snapshot is taken at the start of the transaction and remains consistent until the end of the transaction.
- Transaction-level read consistency eliminates dirty reads, non-repeatable reads, and phantom reads.
- If update conflicts are detected, a participating transaction will roll back.
- Supported on FILESTREAM enabled databases.
- The ALLOW_SNAPSHOT_ISOLATION database option must be ON before you can use the SNAPSHOT isolation level (OFF by default in SQL Server installations, ON by default in Azure SQL Database).

For more information on transaction isolation levels, see the topic *SET TRANSACTION ISOLATION LEVEL (Transact-SQL)* in Microsoft Docs:

 **SET TRANSACTION ISOLATION LEVEL (Transact-SQL)**

<http://aka.ms/faim9a>

Working with Row Versioning Isolation Levels

Row versioning isolation levels (SNAPSHOT isolation, and READ COMMITTED isolation with READ_COMMITTED_SNAPSHOT ON) have costs as well as benefits. In particular, row versioning makes use of **tempdb** to hold versioning data; you should ensure your storage subsystem can accommodate the additional load on **tempdb** before enabling a row versioning isolation level.

In general, row versioning based isolation levels have the following benefits:

- Readers do not block writers.
- Fewer locks overall—SELECT statements do not acquire locks:
 - Reduced blocking and deadlocks
 - Fewer lock escalations

Row versioning-based isolation levels can cause the following issues:

- Read performance may degrade because the set of versioned rows ages and large version chains must be scanned.
- Increased resource utilization in maintaining row versions in **tempdb**.
- Versions are maintained even when there is no active transaction using a row versioning isolation level.
- SNAPSHOT isolation may result in transaction rollback because of update conflict. Applications may need to be modified to handle update conflicts.

Other points you should consider include:

- SNAPSHOT isolation does not affect queries with lock hints. The lock hints still apply.
- Writers still block writers.
- Setting READ_COMMITTED_SNAPSHOT ON requires that only one connection is open (the connection issuing the command). This can be challenging in a production database.
- When READ_COMMITTED_SNAPSHOT is ON, row versioning may be bypassed by using the READCOMMITTEDLOCK table hint, in which case the table will not be row versioned for the purposes of the statement using the hint.

- Row versioning benefits:
 - Fewer locks
 - Less blocking
- Row versioning issues:
 - Versioning is expensive and adds load to **tempdb**
 - Applications need to handle update conflicts
- Other considerations:
 - Lock hints still apply
 - Writers still block writers
 - Setting READ_COMMITTED_SNAPSHOT ON requires a single active connection

Transactions

A transaction is considered to be one or more Transact-SQL statements that are logically grouped into a single unit of work. A transaction might be made up of multiple statements; changes made to data by these statements are applied to the database only when the transaction completes successfully. A transaction must adhere to the ACID principles:

- **Atomicity.** A transaction must be atomic in nature; that is, either all of the changes are applied or none.
- **Consistency.** After completion, a transaction must leave all data and related structures in a consistent state.
- **Isolation.** A transaction must have a view of the data independent of any other concurrent transaction; a transaction should not see data in an intermediate state.
- **Durability.** Data changes must be persisted in case of a system failure.

- A logical unit of work, made up of one or more Transact-SQL statements
 - Atomicity
 - Consistency
 - Isolation
 - Durability
- Transaction management modes:
 - Auto-commit
 - Explicit transactions
 - Implicit transactions
 - Batch-scoped transactions

SQL Server Transaction Management Modes

Auto-commit mode

Auto-commit is the default transaction management mode in SQL Server. A transaction is either committed or rolled back after completion. If a statement completes successfully without any error, it is committed. If it encounters errors, it is rolled back. Auto-commit mode is overridden when a user initiates an explicit transaction or when implicit transaction mode is enabled.

Explicit transaction mode

In explicit transaction mode, you explicitly define the start and end of a transaction.

- BEGIN TRANSACTION. Marks the start of a transaction.
- COMMIT TRANSACTION. Marks the successful completion of a transaction. The modifications made to a database are made permanent; the resources held by a transaction are released.
- ROLLBACK. Marks the unsuccessful termination of a transaction; the modifications made by a transaction are discarded; the resources held by a transaction are released.

When an explicit transaction completes, the connection returns to the transaction mode it was using before the start of the explicit transaction.

For more information on transaction control statements in explicit transaction mode, see the topic *Transaction Statements (Transact-SQL)* in Microsoft Docs:

Transaction Statements (Transact-SQL)

<http://aka.ms/krt2iy>

Implicit transaction mode

In implicit transaction mode, SQL Server automatically manages the start of a transaction. You can commit or roll back an implicit transaction but you cannot control the start of the transaction. SQL Server automatically starts a new implicit transaction after the current implicit transaction finishes, generating a continuous chain of transactions.

- Implicit transaction is a session level setting and can be changed by setting the IMPLICIT_TRANSACTION option to ON/OFF.
- SQL Server automatically starts an implicit transaction when any of the following statements are executed: ALTER TABLE, CREATE, DELETE, DROP, FETCH, GRANT, INSERT, OPEN, REVOKE, SELECT, TRUNCATE TABLE, UPDATE.

For more information about implicit transaction mode, see the topic *SET IMPLICIT_TRANSACTIONS (Transact-SQL)* in Microsoft Docs:

SET IMPLICIT_TRANSACTIONS (Transact-SQL)

<http://aka.ms/vima93>

Batch-scoped transaction mode

The batch-scoped transaction is applicable only to multiple active result sets (MARS). A transaction (whether implicit or explicit) that starts under MARS is converted to a batch-scoped transaction. A batch-scoped transaction that is neither committed nor rolled back on batch completion is automatically rolled back by SQL Server.

The transaction mode is set at the connection level. If a connection changes from one transaction mode to another, other active connections are not affected.

Working with Transactions

You should be aware of some features of transactions in SQL Server when you start to use them.

Naming Transactions

Transaction names are optional, and have no effect on the behavior of SQL Server; they act purely as labels to assist developers and DBAs in understanding Transact-SQL code.

Explicit transactions may be named.

- Naming Transactions:
 - Label only; no effect on code
- Nesting Transactions:
 - Only the state of the outer transaction has any effect
 - @@TRANCOUNT track transaction nesting
- Terminating Transactions:
 - Resource error
 - SET XACT_ABORT
 - Connection closure
- Transaction Best Practices:
 - Keep transactions as short as possible

Transaction Naming Example

```
BEGIN TRANSACTION my_tran_name;
COMMIT TRANSACTION my_tran_name;
```

Nesting Transactions

Explicit transactions can be nested; you can issue a BEGIN TRANSACTION command inside an open transaction. Only the outermost transaction has any effect; if the outermost transaction is committed, all the nested transactions are committed. If the outermost transaction is rolled back, all the nested transactions are rolled back. The level of transaction nesting is tracked by the @@TRANCOUNT function, which is maintained at connection level:

- Each BEGIN TRANSACTION statement increments @@TRANCOUNT by one.
- Each COMMIT statement decrements @@TRANCOUNT by one. The COMMIT that reduces @@TRANCOUNT to zero commits the outermost transaction.
- A ROLLBACK statement rolls back the outer transaction and reduces @@TRANCOUNT to zero.

Terminating Transactions

In addition to an explicit COMMIT or ROLLBACK, a transaction can be terminated for the following reasons:

- **Resource error.** If a transaction fails because of a resource error such as lack of disk space, SQL Server automatically rolls back the transaction to maintain data integrity.
- **SET XACT_ABORT.** When the connection-level SET XACT_ABORT setting is ON, an open transaction is automatically rolled back in the event of a runtime error. When XACT_ABORT is OFF, a statement that causes an error is normally rolled back, but any open transaction will remain open.
- **Connection closure.** If a connection is closed, all open transactions are rolled back.

For more information on SET XACT_ABORT, see the topic *SET XACT_ABORT (Transact-SQL)* in Microsoft Docs:

SET XACT_ABORT (Transact-SQL)

<http://aka.ms/nrph4c>

Savepoints

You can also create a save point partway through the transaction. This is a named point within a transaction. You then use a ROLLBACK statement to revert to the save point, which means you can retry without having to start the entire transaction again. You set save points using the SAVE TRANSACTION key words. You can specify a name up to 32 characters, or a variable holding the name of the save point. You can then use the ROLLBACK command, together with the name of the save point, to return to the save point in the event of an error. Typically the save point would be created in the TRY block, and the ROLLBACK to the save point would be included in the CATCH block of an error handling construct.

The following code fragments show how to name a save point, and how to refer to the named save point in the event of a ROLLBACK:

SAVEPOINT

```
SAVE TRANSACTION SavePointName;  
  
ROLLBACK TRANSACTION ProcedureSave;
```

For more information about SAVE TRANSACTION, see Microsoft Docs:

SAVE TRANSACTION (Transact-SQL)

<https://aka.ms/ypdjku>

Transaction Best Practices

- **Short transactions.** Keep transactions as short as possible. The shorter the transaction, the sooner the locks will be released. This will help reduce unnecessary blocking.
- **Avoid user input.** Avoid user interaction during a transaction. This might add unnecessary delay, because a user may open a transaction and go out for a break. The transaction will hold locks until the user returns to complete the transaction or the transaction is killed. Other transactions requiring locks on the same resource will be blocked during this time.
- **Open a transaction only when necessary.** If possible, avoid opening a transaction when browsing through data. Do the preliminary data analysis and then open a transaction to perform any necessary data modification.

- **Access only relevant data.** Access only the relevant data within a transaction. This reduces the number of locks, so reducing the blocking and deadlocks.
- **Use the appropriate transaction isolation level.** Not all applications require high level isolation level, such as repeatable read and serializable. Many applications work well with the default repeatable read isolation.
- **Beware of triggers containing transactions.** Triggers containing transactions should be written carefully. Issuing a ROLLBACK command within a trigger will roll back the whole transaction, of which the trigger is a part.

Demonstration: Analyzing Concurrency Problems

In this demonstration, you will see:

- Examples of concurrency problems.
- How changes to transaction isolation levels address concurrency problems.

Demonstration Steps

Preparation

1. Ensure that the **MT17B-WS2016-NAT**, **20762C-MIA-DC**, and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Demofiles\Mod17** folder as Administrator.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.
4. Start SQL Server Management Studio and connect to your Azure instance of the **AdventureWorksLT** database engine instance using SQL Server authentication.
5. In the **Login** box, type **Student**, in the **Password** box, type **Pa55w.rd**, and then click **Connect**.
6. Open the **Demo1.ssmssln** solution in the **D:\Demofiles\Mod17\Demo1** folder.
7. Open the **Demo 1a.sql** and the **Demo 1b.sql** script files; open these files in different query windows, because you will be switching between them.
8. In both script files, in the **Available databases** list, click **ADVENTUREWORKSLT**.

Dirty Reads

1. In the **Demo 1a.sql** script file, under the comment that begins **Step 1**, select the code, and then click **Execute** to check the current settings for SNAPSHOT isolation.
2. Under the comment that begins **Step 2**, select the code, and then click **Execute** to view the current state of the row used in this demonstration.
3. In the **Demo 1b.sql** script file, under the comment that begins **Query 1**, select the code, and then click **Execute**.
4. In the **Demo 1a.sql** file, under the comment that begins **Step 3**, select the code, and then click **Execute** to demonstrate READ UNCOMMITTED isolation.
5. Under the comment that begins **Step 4**, select the code, and then click **Execute** to demonstrate READ COMMITTED isolation. The query will wait until you complete the next step.
6. In the **Demo 1b.sql** file, under the comment that begins **Query 2**, select the code, and then click **Execute**.

7. In the **Demo 1a.sql** script file, note that the query under **Step 4** (which was already running) has now returned a result.

Non-repeatable Reads

1. In the **Demo 1a.sql** script file, under the comment that begins **Step 5**, select the first five lines of code, and then click **Execute**.
2. In the **Demo 1b.sql** file, under the comment that begins **Query 3**, select the code, and then click **Execute**.
3. In the **Demo 1a.sql** file, under the comment that begins **Step 5**, select the final four lines of code, and then click **Execute** to demonstrate a non-repeatable read.
4. Under the comment that begins **Step 6**, select the first six lines of code, and then click **Execute**.
5. In the **Demo 1b.sql** file, under the comment that begins **Query 4**, select the code, and then click **Execute**. Note that this query will not return until you complete the next step, because REPEATABLE READ holds a lock on the affected row.
6. In the **Demo 1a.sql** script file, under the comment that begins **Step 6**, select the final four lines of code, and then click **Execute** to demonstrate that REPEATABLE READ isolation prevents a non-repeatable read.

Phantom Read

1. In the script **Demo1a.sql** that begins **Step 7**, select the first six lines of code, and then click **Execute**.
2. In the **Demo 1b.sql** script file, under the comment that begins **Query 5**, select the code, and then click **Execute**.
3. In the **Demo 1a.sql** file, under the comment that begins **Step 7**, select the final four lines of code, and then click **Execute** to demonstrate that READ COMMITTED isolation allows a phantom read.

Serializable and Phantom Read

1. In the script **Demo1a.sql** under the comment that begins **Step 8**, select the first six lines of code, and then click **Execute**.
2. In the **Demo 1b.sql** script file under the comment that begins **Query 6**, select the code, and then click **Execute**. Note that this query will not return until you complete the next step, since SERIALIZABLE holds a lock on the affected table.
3. In the **Demo 1a.sql** script file, under the comment that begins **Step 8**, select the final four lines of code, and then click **Execute** to demonstrate that SERIALIZABLE isolation prevents a phantom read.

Snapshot Isolation

1. In **Demo 1a.sql**, under the comment that begins **Step 9**, select the first eight lines of code, and then click **Execute**. This can take up to 30 minutes to complete.
2. Select the last three lines of code under **Step 9**, and then click **Execute**. Note the locks.
3. In the **Demo 1b.sql** script file, under the comment that begins **Step 6**, select the code, and then click **Execute**. Note that the query does not complete.
4. In the **Demo 1a.sql** script file, under the comment that begins **Query 10**, select the code, and then click **Execute**. Note the locks.
5. In the **Demo 1a.sql** script file, under the comment that begins **Step 11**, select the code, and then click **Execute**. Note that the query in Demo 1b aborted, demonstrating the behavior of SNAPSHOT isolation.
6. Close SQL Server Management Studio without saving any changes.

Check Your Knowledge

Question
User A starts to update a customer record, and while the transaction is still in progress, User B tries to update the same record. User A's update completes successfully, but User B's update fails with an error message: "This customer's record has been updated by another user". Which concurrency model is the system using?
Select the correct answer.
Pessimistic concurrency
Optimistic concurrency

Lesson 2

Locking Internals

SQL Server uses locks to ensure the consistency of data during a transaction. This lesson discusses the details of the locking architecture used by SQL Server, how locks are used during the life of a transaction, and the various methods available to you to influence the default locking behavior.

Lesson Objectives

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

- Describe the SQL Server locking architecture.
- Describe lock hierarchy and lock granularity.
- Explain lock escalation.
- Understand lock modes.
- Explain lock compatibility.
- Explain the data modification process.
- Use locking hints.
- Understand deadlocks.
- Explain latches and spinlocks.

Locking Architecture

In a hypothetical database system, the least sophisticated locking architecture possible is to allow locks only at database level. Every user reading or writing data would lock the entire database, preventing access by any other user until the change was complete. While this approach ensures data is consistent, it prohibits concurrent database activity.

SQL Server's locking system is designed to ensure data consistency while still allowing concurrent activity. Locks are acquired at an appropriate level of granularity to protect the data that is modified by a transaction; locks are held until the transaction commits or rolls back. Different objects affected by a transaction can be locked with different types of lock.

• Locking architecture is designed as a balance between consistency and concurrency:

- Locks—logical level
- Latches—physical level

• Lock manager manages locks:

- Each lock has a lock block
- Each lock block has one or more lock owner blocks
- Lock hash table maintained for better performance

Locks and Latches

SQL Server implements two locking systems. The first system manages locks for database objects (tables, indexes, and so on) that are accessible directly to users; these locks act at a logical level to ensure data consistency. This locking system is managed by the lock manager. The second system is used to ensure the physical consistency of data in memory; for this process, a lightweight locking mechanism, known as a latch, is employed. This system is managed by the storage engine.

Lock Manager

Internally, locks are automatically managed by the lock manager, a component of the database engine. When the database engine processes a Transact-SQL statement, the Query Processor subcomponent determines the resources that will be accessed. The Query Processor also determines the type of locks to acquire, based on the type of data access (read or write) and the transaction isolation level setting. The Query Processor then requests these locks from the lock manager. The lock manager grants the locks if no conflicting locks are being held by other transactions.

Locks are in-memory structures; the lock manager maintains a memory structure for each lock, called a *lock block*, which records the lock type and the resource that is locked. Each lock block will be linked to one or more *lock owner blocks*; the lock owner block links a lock to the process requesting the lock. The lock manager also maintains a lock hash table, to track locked resources more efficiently.

When a SQL Server instance starts, lock manager acquires sufficient memory to support 2,500 locks. If the total number of locks exceeds 2,500, more memory is allocated dynamically to lock manager.

For more information on locking in SQL Server, see the topic *SQL Server Transaction Locking and Row Versioning Guide* on MSDN:

 **SQL Server Transaction Locking and Row Versioning Guide**

<http://aka.ms/mc5pmh>

Lock Granularity and Hierarchy

Database objects and resources can be locked at different levels of granularity; to allow more concurrent activity, SQL Server will attempt to lock as few resources as possible to efficiently process a Transact-SQL statement. The efficiency of the locking strategy is determined by comparing the overhead of maintaining many locks at a fine grain against the increase in concurrency from lower-grained locking. Locking at higher granularity levels—such as at table level—decreases concurrency because the entire table is inaccessible to other transactions. However, the overhead is less, as fewer locks are to be maintained.

- RID, KEY
- PAGE
- EXTENT
- HoBT
- OBJECT
- FILE
- APPLICATION
- METADATA
- ALLOCATION_UNIT
- DATABASE

SQL Server acquires locks at any of the following levels, ordered here from lowest grain to highest grain. The first two items (RID and KEY) are of equivalent grain:

- **RID.** RID stands for row id. A row id is a lock on a single row in a heap (table without clustered index).
- **KEY.** A key lock applies to a single row in a clustered or nonclustered index.
- **PAGE.** A lock on an 8 KB page in a database, such as a data or index page. If a page is locked, all of the data rows contained in the page are locked.
- **EXTENT.** A lock on a 64 KB extent (a block of eight pages). If an extent is locked, all of the pages in the extent are locked.
- **HoBT.** HoBT stands for heap or b-tree. A lock protecting a b-tree (index), or the heap data pages in a table that does not have a clustered index. All the extents that make up the heap or b-tree are locked.
- **OBJECT.** Typically, a lock on a table. If a table is locked, all of the associated data and index pages are also locked.

- **FILE**. A lock on a database file. If a file is locked, all of the objects it contains are locked.
- **APPLICATION**. An application-specified lock, created using **sp_getapplock**.
- **METADATA**. A lock on catalog views.
- **ALLOCATION_UNIT**. A lock on an allocation unit such as IN_ROW_DATA.
- **DATABASE**. A lock on an entire database. All the objects in the database are also locked.

The objects in this list make up a hierarchy; databases are composed of files, files contain tables, and tables are in turn made up of extents, pages, and rows. To fully protect a resource during the processing of a command, a process might acquire locks at multiple levels of the resource hierarchy. For example, when processing a command that affects a single row in a table, locks might be acquired on the affected row, the page in which the row is stored, the page's extent, and the table to which the row belongs. This both fully protects the table and simplifies the detection of locking conflicts with other concurrent processes that may hold locks on different rows in the same table.

Lock Escalation

Lock escalation occurs when many fine-grained locks held by a transaction on a single resource are converted to a single coarser-grained lock on the same resource. Lock escalation is used to limit the total number of locks the lock manager must manage; the cost being that lock escalation might reduce concurrency.

When lock escalation from row locks occurs, the lock is always escalated to table level; lock escalation does not take place from row level to page level.

In previous versions of SQL Server, the default conditions for lock escalation were hard-coded; when a transaction held more than a fixed number of row level or page level locks on a resource, the lock would be escalated. This is no longer true and lock escalation decisions are now based on multiple factors; there is no fixed threshold for lock escalation.

Lock escalation can also occur when the memory structures maintained by the lock manager consume more than 40 percent of the available memory.

You can control lock escalation behavior for individual tables by using the ALTER TABLE SET LOCK_ESCALATION command. LOCK_ESCALATION can be set to one of three values:

- **TABLE**. The default value. When lock escalation occurs, locks are always escalated to table level, whether or not the table is partitioned.
- **AUTO**. If the table is partitioned when lock escalation occurs, locks can be escalated to partition level. If the table is not partitioned, locks are escalated to table level.
- **DISABLE**. Prevents lock escalation occurring in most cases. Table locks might still occur, but will be less frequent.

- Reduces lock manager memory overhead by converting many fine-grained locks to a single coarser-grained lock
 - Row and page locks escalate to table locks
- Control at table level with ALTER TABLE SET LOCK_ESCALATION
- Control at session or instance level with trace flags 1224 and 1211

For more information on controlling lock escalation behavior, see the topic *ALTER TABLE (Transact-SQL)* in Microsoft Docs:



ALTER TABLE (Transact-SQL)

<http://aka.ms/hb1ub7>

Lock escalation behavior can also be controlled at session level or instance level by use of trace flags:

- Trace flag 1224 disables lock escalation, based on the number of locks held on a resource. Lock escalation due to memory pressure can still occur.
- Trace flag 1211 disables lock escalation completely, whether due to the number of locks held on a resource or due to memory pressure. Disabling lock escalation can have a severe effect on performance and is not recommended.

For more information on trace flags 1224 and 1211, see the topic *Trace Flags (Transact-SQL)* in Microsoft Docs:

Trace Flags (Transact-SQL)

<http://aka.ms/hvmsq7>

Lock Modes

SQL Server locks resources using different lock modes. The lock modes determine how accessible a resource is to other concurrent transactions.

Data Lock Modes

The following lock modes are used to lock resources:

- **Shared lock.** Shared locks are acquired when reading data. The duration for which a shared lock is held depends on transaction isolation level or locking hints. Many concurrent transactions might hold shared locks on the same data. No other transaction can modify the data until the shared lock is released.
- **Exclusive lock.** Exclusive locks are acquired when data is modified (by an INSERT, UPDATE, or DELETE statement). Exclusive locks are always held until the end of the transaction. Only one transaction may acquire an exclusive lock on a data item at a time; while an exclusive lock is held on a data item, no other type of lock may be acquired on that data item.
- **Update lock.** Update locks are acquired when modifying data and are a combination of shared and exclusive locks. Update locks are held during the searching phase of the update, where the rows to be modified are identified; they are converted to exclusive locks when actual modification takes place. Only one transaction may acquire an update lock on a data item at one time; other transactions might hold or acquire shared locks on the same data item while an update lock is in place.
- **Intent lock.** An intent lock is not a locking mode in its own right—it acts as a qualifier to other lock modes. Intent locks are used on a data item to indicate that a subcomponent of the data item is locked; for instance, if a row in a table is locked with a shared lock, the table to which the row belongs would be locked with an intent shared lock. Intent locks are discussed in more detail in the next topic.
- **Key-range locks.** Key-range locks are used by transactions using the SERIALIZABLE isolation level to lock ranges of rows that are implicitly read by the transaction; they protect against phantom reads.

• Data Lock Modes:

- Shared
- Exclusive
- Update
- Intent
- Key-range

• Special Lock Modes:

- Schema
- Conversion
- Bulk update

• Lock mode names are abbreviated in DMVs

Special Lock Modes

Special lock modes are used to control stability of the database schema, when locks are converted between modes, and during bulk update operations:

- **Schema lock.** Schema locks are used when an operation dependent on the table schema is executed. There are two types of schema lock:
 - **Schema modification lock.** Schema modification locks are acquired when a data definition language (DDL) operation is being performed against a table, such as adding or dropping a column.
 - **Schema stability lock.** Schema stability locks are used during query compilation to prevent transactions that modify the underlying database schema. Schema stability locks are compatible with all other lock types (including exclusive locks).
- **Conversion lock.** A conversion lock is a specialized type of intent lock used to manage the transition between data lock modes. Conversion locks appear in three types:
 - **Shared with intent exclusive.** Used when a transaction holds a mixture of shared locks and exclusive locks on subobjects of the locked object.
 - **Shared with intent update.** Used when a transaction holds a mixture of shared locks and update locks on subobjects of the locked object.
 - **Update with intent exclusive.** Used when a transaction holds a mixture of exclusive locks and update locks on subobjects of the locked object.
- **Bulk update lock.** Bulk update locks can optionally be acquired when data is bulk inserted into a table using a bulk command such as BULK INSERT. A bulk update lock can only be acquired if no other incompatible lock types are held on the table.

Locks held by active transactions can be viewed by using the **sys.dm_tran_locks** dynamic management view (DMV). DMVs use abbreviations for lock mode names, summarized in the following table:

Abbreviation	Lock Mode
S	Shared
U	Update
X	Exclusive
IS	Intent Shared
IU	Intent Update
IX	Intent Exclusive
RangeS_S	Shared Key-Range and Shared Resource lock
RangeS_U	Shared Key-Range and Update Resource lock
Rangel_N	Insert Key-Range and Null Resource lock
Rangel_S	Key-Range Conversion lock
Rangel_U	Key-Range Conversion lock
Rangel_X	Key-Range Conversion lock
RangeX_S	Key-Range Conversion lock
RangeX_U	Key-Range Conversion lock

Abbreviation	Lock Mode
RangeX_X	Exclusive Key-Range and Exclusive Resource lock
Sch-S	Schema stability
Sch-M	Schema modification
SIU	Shared Intent Update
SIX	Shared Intent Exclusive
UIX	Update Intent Exclusive
BU	Bulk Update

For more information on the **sys.dm_tran_locks** DMV, see the topic *sys.dm_tran_locks (Transact-SQL)* in Microsoft Docs:

 **sys.dm_tran_locks (Transact-SQL)**

<http://aka.ms/jf98cd>

Lock Mode Compatibility

Processes may acquire locks of different modes. Two lock modes are said to be compatible if a process can acquire a lock mode on a resource when another concurrent process already has a lock on the same resource. If a process attempts to acquire a lock mode that is incompatible with the mode of an existing lock, the process must wait for the existing lock to be released before acquiring the new lock.

SQL Server uses lock compatibility to ensure transactional consistency and isolation, while still permitting concurrent activity; it allows processes that read data to run concurrently, while ensuring that modification of a resource can only be carried out by one process at a time.

Requested mode	Existing granted mode					
	IS	S	U	IX	SIX	X
Intent shared (IS)	Yes	Yes	Yes	Yes	Yes	No
Shared (S)	Yes	Yes	Yes	No	No	No
Update (U)	Yes	Yes	No	No	No	No
Intent exclusive (IX)	Yes	No	No	Yes	No	No
Shared with intent exclusive (SIX)	Yes	No	No	No	No	No
Exclusive (X)	No	No	No	No	No	No

 **Note:** Lock compatibility gives an insight into the differences in behavior between the different isolation levels you learned about in the previous topic; the more pessimistic isolation levels acquire and hold locks that are less compatible with other lock types.

When processes wait for incompatible lock types to be released, they wait in a first-in, first-out queue. If there are already processes queuing for a resource, a process seeking to acquire a lock on the same resource must join the end of the queue, even if the mode of the lock it seeks to acquire is compatible with the current lock on the resource. On busy resources, this prevents processes seeking less compatible lock modes from waiting indefinitely when other, more compatible, lock modes are in use.



Note: When a process is waiting for an incompatible lock on a resource to be released, it is said to be blocked. Because of the way processes queue when waiting for locks, chains of blocked processes can develop, slowing—or potentially stopping—system activity.

For a full lock compatibility matrix, see the topic *Lock Compatibility (Database Engine)* on Microsoft TechNet (note that this page comes from the SQL Server 2008 R2 documentation; Microsoft has not published a recent version of this matrix):



Lock Compatibility (Database Engine)

<http://aka.ms/t0ia22>

The Data Modification Process

To understand how locks are used as Transact-SQL statements are processed, consider the example of a statement that modifies data in an UPDATE statement.

The following UPDATE query could be run in the AdventureWorks database:

Example UPDATE Statement Changing Two Rows

```
UPDATE HumanResources.Employee
SET MaritalStatus = 'S'
WHERE BusinessEntityId IN (3,289);
```

- Relevant data pages located in the Buffer Pool
- Locks before data modification:
 - Update lock on affected rows
 - Intent exclusive lock on pages table
 - Intent exclusive lock on table
 - Shared lock on database
- Data modification locks:
 - Update lock converted to an exclusive lock on affected rows
 - Intent exclusive lock on pages table
 - Intent exclusive lock on table
 - Shared lock on database

The following steps are involved when modifying data in SQL Server:

- A user or an application sends the UPDATE query to SQL Server.
- The database engine receives the update request and locates the data pages to be updated in the cache—or reads the data pages from the storage subsystem into cache.
- The database engine tries to grant the lock on the necessary data to the user's session:
 - If any transaction already has an incompatible lock on the affected data, the UPDATE query waits for the existing lock to be released.
 - Because this UPDATE statement is highly selective (affecting only two rows) the database engine uses row level locking to acquire an update lock on each of the two rows being modified.
- The following additional locks are acquired to secure the pages and the table in question:
 - Two intent-exclusive page level locks (one for each page, since the rows are in different pages).
 - One intent-exclusive table level lock.
 - A shared database level lock.
 - Additional locks may be required if the data being modified makes up an index. In this example, no indexes are affected.
- SQL Server starts the data modification. The steps are as follows:
 - The data modification is made (in the cache). At the same time, the update lock is converted to an exclusive lock.

- The changes are logged in transaction log pages (in cache).
- The locks are released.
- The transaction is committed.
- Acknowledgement is sent to the user or application.

Locking Hints

There might be instances where you need to influence locking behavior; several table hints are available that help you adjust the locking of individual tables during a single Transact-SQL statement. You can use hints to influence:

- The mode of any locks acquired on a table.
- The transaction isolation level applied to a table.

Table hints are applied by including a WITH command after the name of the table for which you want to influence locking in the FROM clause of a Transact-SQL statement.

Table Hint Example

```
...  
FROM <table name> WITH (<table hint> [,<table hint>])
```

You can specify multiple hints for a table—the hints should be comma-separated in the brackets of the WITH command.



Best Practice: In general, it is best to avoid locking hints and allow the SQL Server Query Optimizer to select an appropriate locking strategy. Be sure to regularly review any locking hints you use; confirm that they are still appropriate.

Hints Affecting Lock Mode

The following hints affect the lock mode acquired by a Transact-SQL statement:

- ROWLOCK. Row locks should be acquired where page or table locks would normally be used.
- PAGLOCK. Page locks should be acquired where row or table locks would normally be used.
- TABLOCK. A table lock should be acquired where row or page locks would normally be used.
- TABLOCKX. An exclusive table lock should be acquired.
- UPDLOCK. An update lock should be acquired.
- XLOCK. An exclusive lock should be acquired.

Hints Affecting Table Isolation Level

The following hints affect the isolation level used by a Transact-SQL statement:

- READCOMMITTED. Use the READ COMMITTED isolation level. Locks or row versioning are used, depending on the value of the READ_COMMITTED_SNAPSHOT database setting.

• Hints Affecting Lock Mode:

- ROWLOCK
- PAGLOCK
- TABLOCK
- TABLOCKX
- UPDLOCK
- XLOCK

• Hints Affecting Table Isolation Level:

- READCOMMITTED
- READCOMMITTEDLOCK
- READUNCOMMITTED or NOLOCK
- REPEATABLEREAD
- SERIALIZABLE or HOLDLOCK
- READPAST

- READCOMMITTEDLOCK. Use the READ COMMITTED isolation level, acquiring locks. The value of the READ_COMMITTED_SNAPSHOT database setting is ignored.
- READUNCOMMITTED or NOLOCK. Use the READ UNCOMMITTED isolation level. Both READUNCOMMITTED and NOLOCK hints have the same effect.
- REPEATABLEREAD. Use the REPEATABLE READ isolation level.
- SERIALIZABLE or HOLDLOCK. Use the SERIALIZABLE isolation level. Both SERIALIZABLE and HOLDLOCK hints have the same effect.
- READPAST. Rows that are locked by other transactions will be ignored, instead of waiting for incompatible locks to be released,

For more information on table hints—including those that control locking—see the topic *Table Hints (Transact-SQL)* in Microsoft Docs:

Table Hints (Transact-SQL)

<http://aka.ms/fkaztl>

Some best practices when using locking hints are:

- Use the TABLOCK hint to speed up bulk insert operations. TABLOCK is only compatible with itself. This allows multiple bulk inserts to be made in parallel into a single table, while preventing other processes to update or modify the records. This considerably improves bulk insert performance.
- Avoid using the NOLOCK or READUNCOMMITTED hint to resolve reader-writer blocking; consider setting READ_COMMITTED_SNAPSHOT to ON or using the SNAPSHOT isolation level. The NOLOCK and READUNCOMMITTED hints are only suitable in environments where the effects of the READ UNCOMMITTED isolation level (documented in the previous lesson) are acceptable.
- Use ROWLOCK or UPDLOCK hints to reduce deadlocks in the REPEATABLE READ isolation level.

Deadlock Internals

A deadlock occurs when two or more transactions block one another by attempting to acquire a lock on a resource that is already locked by the other transaction(s) with an incompatible lock mode. For example:

- Transaction A acquires a shared lock on table T1.
- Transaction B acquires a shared lock on table T2.
- Transaction A requests an exclusive lock on table T2. It waits on transaction B to release the shared lock it holds on table T2.
- Transaction B requests an exclusive lock on table T1. It waits on transaction A to release the shared lock it holds on Table T1. A deadlock has occurred.

- Deadlocks are resolved by the Lock Manager:
 - Runs every five seconds by default; frequency increases as deadlocks are detected
 - Deadlock victim is selected and terminated
 - Deadlock priority can be used to influence the likelihood that a transaction will be selected as the deadlock victim
 - Use SQL Server Profiler to analyze deadlocks
 - Deadlock graphs

Without intervention, a deadlock will continue indefinitely.

Deadlock Resolution

The Lock Monitor process is responsible for detecting deadlocks. It periodically searches for the tasks involved in a deadlock. The search process has the following properties:

- The default interval between deadlock detection searches for five seconds.
- As soon as a deadlock is found, the deadlock detection search will run again immediately.
- When deadlocks are detected, the deadlock detection search interval is reduced to as little as 100 milliseconds, depending on the deadlock frequency.

When a deadlock is detected, the Lock Monitor ends the deadlock by choosing one of the threads as the deadlock victim. The deadlock victim command is forcefully terminated; the transaction is rolled back, and the error 1205 is returned to the application. This releases the locks held by the deadlock victim, allowing the other transactions to continue with their work.

The deadlock victim is selected, based on the following rules:

- If all the deadlocked transactions have the same deadlock priority, the transaction that is estimated to be the least expensive to roll back is chosen as the deadlock victim.
- If the deadlocked transactions have a different deadlock priority, the transaction with the lowest deadlock priority is chosen as the deadlock victim.

Deadlock Priority

You can specify the deadlock priority of a transaction by using the `SET DEADLOCK_PRIORITY` command.

You can set deadlock priority to an integer value between -10 (the lowest priority) and 10 (the highest priority)—or you can use a text value:

- **LOW**. Equivalent to the integer value -5.
- **NORMAL**. Equivalent to the integer value 0.
- **HIGH**. Equivalent to the integer value 5.

For more information on setting deadlock priority, see the topic *SET DEADLOCK_PRIORITY (Transact-SQL)* in Microsoft Docs:

 **SET DEADLOCK_PRIORITY (Transact-SQL)**

<http://aka.ms/vaffc7>

Analyze Deadlocks

You can use SQL Server Profiler to analyze deadlocks. The **Locks** section of SQL Server Profiler includes a **Deadlock graph** event class which you can add to your trace. This is a graphical representation of the deadlock, showing the statements involved in the deadlock. Add data about the deadlock to a Deadlock XML file, and you can choose whether to write all deadlock events to one file, or to separate files. The deadlock data can then be viewed graphically by opening the file in SSMS.

For more information about analyzing deadlocks using SQL Server Profiler, see Microsoft Docs:

 **Analyze Deadlocks with SQL Server Profiler**

<https://aka.ms/nn6oel>

Latches and Spinlocks

Some database engine operations avoid the cost of managing locks by using lighter-weight locking mechanisms, latches, and spinlocks.

Latches

Latches are a lightweight locking mechanism used by the storage engine to ensure the consistency of in-memory data structures, such as data pages and non-leaf pages in a b-tree. Latches are managed internally by SQL Server and cannot be controlled by users. Latches are broadly divided into three types:

- **I/O latches.** Used to manage outstanding I/O operations against pages in the Buffer Pool, I/O latches ensure that pages are read only once from I/O into the Buffer Pool.
- **Buffer latches.** Used to prevent concurrent processes from making conflicting changes to pages in the Buffer Pool.
- **Non-buffer latches.** Used to protect shared data structures held outside the Buffer Pool.

When a process waits for a latch, the duration of the wait is recorded in the **sys.dm_os_wait_stats** DMV:

- I/O latches appear as wait types with names starting PAGEIOLATCH_.
- Buffer latches appear as wait types with names starting PAGELATCH_.
- Non-buffer latches are summarized as wait types with names starting LATCH_. A complete list of all non-buffer latch types can be found in the **sys.dm_os_latch_stats** DMV.

For more information on the **sys.dm_os_wait_stats** DMV, see the topic *sys.dm_os_wait_stats (Transact-SQL)* in Microsoft Docs:

 [sys.dm_os_wait_stats \(Transact-SQL\)](#)

<http://aka.ms/kvkoru>

For more information on the **sys.dm_os_latch_stats** DMV, see the topic *sys.dm_os_latch_stats (Transact-SQL)* in Microsoft Docs:

 [sys.dm_os_latch_stats \(Transact-SQL\)](#)

<http://aka.ms/im1px3>

Spinlocks

Spinlocks are very lightweight locking structures used when a process needs to lock an object in memory for a very short time. A process waiting to acquire a spinlock will go into a loop for a period, checking repeatedly whether the lock is available—as opposed to moving onto a waiter list and yielding the CPU immediately. SQL Server uses spinlocks to protect objects such as hash buckets in the lock manager's lock hash table.

Some contention for spinlocks is expected on busy SQL Server instances; spinlock contention should only be considered a problem when it causes significant CPU overhead. Performance problems can be caused by contention for spinlocks, but this is a relatively rare occurrence.

- **Latches:**
 - Protect pages in memory
 - I/O latches. PAGEIOLATCH_ waits
 - Buffer latches. PAGELATCH_ waits
 - Non-buffer latches. LATCH_ waits

- **Spinlocks:**
 - Very lightweight locks
 - Rarely cause performance problems

For more information on diagnosing and resolving performance problems caused by spinlock contention, see the Microsoft paper *Diagnosing and Resolving Spinlock Contention on SQL Server*. Note that this paper was written in reference to SQL Server 2008 R2:

 **Diagnosing and Resolving Spinlock Contention on SQL Server**

<http://aka.ms/uvpmoe>

Demonstration: Applying Locking Hints

In this demonstration, you will see the effects of several locking hints.

Demonstration Steps

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
2. Open the **Demo2.ssmssln** solution in the **D:\Demofiles\Mod17\Demo2** folder.
3. Open the **Demo 2a - lock hints 1.sql** and **Demo 2b - lock hints 2.sql** script files. Ensure that both scripts use the **AdventureWorks** database.
4. In the **Demo 2a - lock hints 1.sql** script file, under the comment that begins **Step 3**, select the code, and then click **Execute** to show the current isolation level.
5. Under the comment that begins **Step 4**, select the first three lines of code, and then click **Execute** to demonstrate the locks held by a transaction using READ UNCOMMITTED isolation.
6. Under the comment that begins **Step 4**, select the remaining five lines of code, and then click **Execute**.
7. Under the comment that begins **Step 5**, select the first three lines of code, and then click **Execute** to demonstrate the locks held by a transaction using REPEATABLE READ isolation.
8. Under the comment that begins **Step 5**, select the remaining five lines of code, and then click **Execute**.
9. Under the comment that begins **Step 6**, select the first three lines of code, and then click **Execute** to demonstrate the locks held by a transaction using REPEATABLE READ isolation and a READCOMMITTED locking hint.
10. Under the comment that begins **Step 6**, select the remaining five lines of code, and then click **Execute**.
11. Under the comment that begins **Step 7**, select the first three lines of code, and then click **Execute** to demonstrate the locks held by a transaction using READ COMMITTED isolation and a TABLOCKX locking hint.
12. Under the comment that begins **Step 7**, select the remaining five lines of code, and then click **Execute**.
13. Under the comment that begins **Step 8**, select the first three lines of code, and then click **Execute** to demonstrate the locks held by a transaction using REPEATABLE READ isolation and a TABLOCKX locking hint.
14. Under the comment that begins **Step 8**, select the remaining five lines of code, and then click **Execute**.
15. In the **Demo 2b - lock hints 2.sql** script file, under the comment that begins **Query 1**, select the code, and then click **Execute**.

16. In the **Demo 2a - lock hints 1.sql** script file, under the comment that begins **Step 9**, select the code, and then click **Execute** to demonstrate that the statement waits.
17. Allow the query to wait for a few seconds, and then on the **Query** menu, click **Cancel Executing Query**.
18. Under the comment that begins **Step 10**, select the code, and then click **Execute** to demonstrate the behavior of the READPAST hint.
19. In the **Demo 2b - lock hints 2.sql** script file, under the comment that begins **Query 2**, select the code, and then click **Execute** to close the open transaction.
20. Close SSMS without saving any changes.

Check Your Knowledge

Question	
If a process is attempting to acquire an exclusive row lock, what lock mode will it attempt to acquire on the data page and table that contain the row?	
Select the correct answer.	
	Exclusive (X)
	Shared (S)
	Intent shared (IS)
	Intent exclusive (IX)
	Intent update (IU)

Lab: Concurrency and Transactions

Scenario

You have reviewed statistics for the **AdventureWorks** database and noticed high wait stats for CPU, memory, IO, blocking, and latching. In this lab, you will address blocking wait stats. You will explore workloads that can benefit from snapshot isolation and partition level locking. You will then implement snapshot isolation and partition level locking to reduce overall blocking.

Objectives

After completing this lab, you will be able to:

- Implement the SNAPSHOT isolation level.
- Implement partition level locking.

Estimated Time: 45 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Implement Snapshot Isolation

Scenario

You have reviewed wait statistics for the **AdventureWorks** database and noticed high wait stats for locking, amongst others. In this exercise, you will implement SNAPSHOT isolation to reduce blocking scenarios.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Clear Wait Statistics
3. Run the Workload
4. Capture Lock Wait Statistics
5. Enable SNAPSHOT Isolation
6. Implement SNAPSHOT Isolation
7. Rerun the Workload
8. Capture New Lock Wait Statistics
9. Compare Overall Lock Wait Time

► Task 1: Prepare the Lab Environment

1. Ensure that the **MT17B-WS2016-NAT**, **20762C-MIA-DC**, and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab17\Starter** folder as Administrator.

► **Task 2: Clear Wait Statistics**

1. Start **SQL Server Management Studio** and connect to the **MIA-SQL** instance using Windows authentication, then open the project file **D:\Labfiles\Lab17\Starter\Project\Project.ssmssln** and the script file **Lab Exercise 01 - snapshot isolation.sql**.
2. Execute the query under the comment that begins **Task 1** to clear wait statistics.

► **Task 3: Run the Workload**

- In the **D:\Labfiles\Lab17\Starter** folder, execute **start_load_exercise_01.ps1** with PowerShell™. Wait for the workload to finish before continuing. If a message is displayed asking you to confirm a change in execution policy, type **Y** and then press Enter.

► **Task 4: Capture Lock Wait Statistics**

- In SSMS, amend the query under the comment that begins **Task 3** to capture only lock wait statistics into a temporary table. Hint: lock wait statistics have a **wait_type** that begins "LCK".

► **Task 5: Enable SNAPSHOT Isolation**

- Amend the properties of the **AdventureWorks** database to allow SNAPSHOT isolation.

► **Task 6: Implement SNAPSHOT Isolation**

1. In SSMS Solution Explorer, open the script file **Lab Exercise 01 - stored procedure.sql**.
2. Use the script to modify the stored procedure definition to run under SNAPSHOT isolation.

► **Task 7: Rerun the Workload**

1. In the SSMS query window for **Lab Exercise 01 - snapshot isolation.sql**, rerun the query under the comment that begins **Task 1**.
2. In the **D:\Labfiles\Lab17\Starter** folder, execute **start_load_exercise_01.ps1** with PowerShell. Wait for the workload to finish before continuing.

► **Task 8: Capture New Lock Wait Statistics**

- In SSMS, under the comment that begins **Task 8**, amend the query to capture lock wait statistics into a temporary table called **#task8**.

► **Task 9: Compare Overall Lock Wait Time**

- In the SSMS query window for **Lab Exercise 01 - snapshot isolation.sql**, execute the query under the comment that begins **Task 9**, to compare the total **wait_time_ms** you have captured between the **#task3** and **#task8** temporary tables.

Results: After this exercise, the **AdventureWorks** database will be configured to use the SNAPSHOT isolation level.

Exercise 2: Implement Partition Level Locking

Scenario

You have reviewed statistics for the **AdventureWorks** database and noticed high wait stats for locking, amongst others. In this exercise, you will implement partition level locking to reduce blocking.

The main tasks for this exercise are as follows:

1. Open Activity Monitor
2. Clear Wait Statistics
3. View Lock Waits in Activity Monitor
4. Enable Partition Level Locking
5. Rerun the Workload

► Task 1: Open Activity Monitor

1. In SSMS Object Explorer, open Activity Monitor for the **MIA-SQL** instance.
2. In Activity Monitor, expand the **Resource Waits** section.

► Task 2: Clear Wait Statistics

1. If it is not already open, open the project file **D:\Labfiles\Lab17\Starter\Project\Project.ssmssln**, then open the query file **Lab Exercise 02 - partition isolation.sql**.
2. Execute the code under **Task 2** to clear wait statistics.

► Task 3: View Lock Waits in Activity Monitor

1. In the **D:\Labfiles\Lab17\Starter** folder, execute **start_load_exercise_02.ps1** with PowerShell. Wait for the workload to finish before continuing (it will take a few minutes to complete).
2. Switch to SSMS and to the **MIA-SQL - Activity Monitor** tab. In the **Resource Waits** section, note the value of **Cumulative Wait Time (sec)** for the **Lock** wait type.
3. Close the PowerShell window where the workload was executed.

► Task 4: Enable Partition Level Locking

1. In the **Lab Exercise 02 - partition isolation.sql** query, under the comment that begins **Task 5**, write a query to alter the **Proseware.CampaignResponsePartitioned** table in the **AdventureWorks** database to enable partition level locking.
2. Rerun the query under the comment that begins **Task 2** to clear wait statistics.

► Task 5: Rerun the Workload

1. In the **D:\Labfiles\Lab17\Starter** folder, execute **start_load_exercise_02.ps1** with PowerShell. Wait for the workload to finish before continuing (it will take a few minutes to complete).
2. Return to the **MIA-SQL - Activity Monitor** tab. In the **Resource Waits** section, note the value of **Cumulative Wait Time (sec)** for the **Lock** wait type.
3. Compare this value to the value you noted earlier in the exercise.
4. Close the PowerShell window where the workload was executed.
5. Close SQL Server Management Studio without saving any changes.

Results: After this exercise, the **AdventureWorks** database will use partition level locking.

Check Your Knowledge

Question	
<p>When partition level locking is enabled, what combination of locks will be held by an UPDATE statement that updates all the rows in a single partition? Assume that the partition contains more than 1 million rows.</p>	
<p>Select the correct answer.</p>	
	Database: Shared (S) Table: Exclusive (X)
	Database: Shared (S) Table: Intent Exclusive (IX) Partition: Exclusive (X)
	Database: Shared (S) Table: Exclusive (X) Partition: Exclusive (X)

Module Review and Takeaways

In this module, you have learned about SQL Server's implementation of transactions and concurrency. You have learned how to use transaction isolation levels to control data consistency within a transaction, and the concurrency issues you might expect at each isolation level. You have also learned about how locking is used to implement transaction isolation levels, and how to use lock hints to modify locking behavior.

Review Question(s)

Check Your Knowledge

Question
A transaction is running with the SERIALIZABLE transaction isolation level. The transaction includes a SELECT statement with a single table in the FROM clause; the table is referenced with the READCOMMITTED table hint. Which transaction isolation level applies to the SELECT statement?
Select the correct answer.
SERIALIZABLE
READ UNCOMMITTED
REPEATABLE READ
READ COMMITTED

Module 18

Performance and Monitoring

Contents:

Module Overview	18-1
Lesson 1: Extended Events	18-2
Lesson 2: Working with Extended Events	18-11
Lesson 3: Live Query Statistics	18-20
Lesson 4: Optimize Database File Configuration	18-23
Lesson 5: Metrics	18-27
Lab: Monitoring, Tracing, and Baselining	18-37
Module Review and Takeaways	18-40

Module Overview

This module looks at how to measure and monitor the performance of your SQL Server® databases. The first two lessons look at SQL Server Extended Events, a flexible, lightweight event-handling system built into the Microsoft® SQL Server Database Engine. These lessons focus on the architectural concepts, troubleshooting strategies and usage scenarios.

This module also describes tools and techniques you can use to monitor performance data, and to create baselines.

Objectives

After completing this module, you will be able to:

- Understand Extended Events and how to use them.
- Work with Extended Events.
- Understand Live Query Statistics.
- Optimize the file configuration of your databases.
- Use DMVs and Performance Monitor to create baselines and gather performance metrics.

Lesson 1

Extended Events

SQL Server Extended Events is a flexible, lightweight event-handling system built into the Microsoft SQL Server Database Engine. This lesson focuses on the architectural concepts, troubleshooting strategies, and usage scenarios of Extended Events.

Lesson Objectives

When you have completed this lesson, you will be able to:

- Understand Extended Events.
- Use Extended Events to capture events.

Extended Events, SQL Trace, and SQL Server Profiler

Extended Events, SQL Trace, and SQL Server Profiler are all tools that you can use to monitor SQL Server events.

SQL Trace

SQL Trace is a server-side, event-driven activity monitoring tool; it can capture information about more than 150 event classes. Each event returns data in one or more columns and you can filter column values. You configure the range of events and event data columns in the trace definition. You can also configure the destination for the trace data, a file or a database table, in the trace definition.

SQL Trace is included in SQL Server 7.0 and later versions.

SQL Server Profiler

SQL Server Profiler is a GUI for creating SQL traces and viewing data from them. SQL Server Profiler is included in SQL Server 7.0 and later versions.

As established parts of the SQL Server platform, SQL Server Profiler and SQL Trace are familiar to many SQL Server administrators.

- SQL Trace and SQL Server Profiler are tools for collecting trace information about activity on a SQL Server instance
- Extended Events is the successor to SQL Trace and will eventually replace it
 - SQL Trace and SQL Server Profiler have been marked for deprecation since SQL Server 2012
 - Extended Events is more flexible than SQL Trace
 - Support for new features added since SQL Server 2012
 - Greater flexibility comes with greater complexity

 **Note:** SQL Trace and SQL Server Profiler are marked for deprecation and will be removed in future versions of SQL Server. Extended Events is now the recommended activity tracing tool. Because SQL Trace is marked for deprecation, it does not include event classes for many features added in SQL Server 2012 onwards.

Extended Events

Like SQL Trace, Extended Events is an event-driven activity monitoring tool; however, it attempts to address some of the limitations in the design of SQL Trace by following a loose-coupled design pattern. Events and their targets are not tightly coupled; any event can be bound to any target. This means that data processing and filtering can be carried out independently of data capture. In most cases, this results in Extended Events having a lower performance overhead than an equivalent SQL Trace.

With Extended Events, you can define sophisticated filters on captured data. In addition to using value filters, you can filter events by sampling, and data can be aggregated at the point it is captured. You can manage Extended Events either through a GUI in SQL Server Management Studio (SSMS) or by using Transact-SQL statements.

You can integrate Extended Events with the Event Tracing for Windows (ETW) framework, so that you can monitor SQL Server activity alongside other Windows® components.

Extended Events was introduced in SQL Server 2008; since the deprecation of SQL Trace and SQL Server Profiler was announced with the release of SQL Server 2012, many features introduced in SQL Server 2012, 2014, 2016, and 2017 can only be traced using Extended Events.

However, the additional flexibility of Extended Events comes at the cost of greater complexity.

Extended Events Architecture

The Extended Events engine is a collection of services, running in the database engine, that provide the resources necessary for events to be defined and consumed.

You might find it most helpful to think about Extended Events primarily in terms of the session object. A session defines the Extended Events data that you want to collect, how the data will be filtered, and how the data will be stored for later analysis. Sessions are the top-level object through which you will interact with Extended Events:

- User defines **session**
 - **Session** includes **event**
 - **Event** triggers **action**
 - **Event** is filtered by **predicate**
 - **Session** writes to **target**

- Extended Events engine provides capabilities
 - User defines session
 - Session collects event
 - Event triggers action
 - Event is filtered by predicate
 - Session writes to target
 - A package defines the objects available to a session

A list of sessions is maintained by the Extended Events engine. You can define and modify sessions using Transact-SQL or in SSMS. You can view data collected by active sessions using Transact-SQL—in which case the data is presented in XML format—or using SSMS.

Packages

Packages act as containers for the Extended Events objects and their definitions; a package can expose any of the following object types:

- Events
- Predicates
- Actions
- Targets
- Types
- Maps

- Executables and executable modules expose Extended Events packages
- A package is a container for other object types:
 - Events
 - Predicates
 - Actions
 - Targets
 - Maps
 - Types

Packages are contained in a module that exposes them to the Extended Events engine. A module can contain one or more packages, and can be compiled as an executable or DLL file.

A complete list of packages registered on the server can be viewed using the **sys.dm_xe_packages** DMV:

sys.dm_xe_packages

```
SELECT * FROM sys.dm_xe_packages;
```

For more information about **sys.dm_xe_packages**, see Microsoft Docs:

 **sys.dm_xe_packages (Transact-SQL)**

<http://aka.ms/i4j6vf>

Events, Actions, and Predicates

Events

Events are points in the code of a module that are of interest for logging purposes. When an event fires, it indicates that the corresponding point in the code was reached. Each event type returns information in a well-defined schema when it occurs.

All available events can be viewed in the **sys.dm_xe_objects** DMV under the **event** object_type.

- Events are logging points in executable code
- When an event fires, it indicates that the associated code has been executed
- Returns data in a fixed schema
- Events are compatible with Event Tracing for Windows
- Use predicates to apply rules to filter events
 - Comparisons—logical operators (=, <, > and so on)
 - Sources—inputs for comparisons
 - Complex predicates may be constructed every *n* events, or every *n* seconds

sys.dm_xe_objects; events

```
SELECT * FROM sys.dm_xe_objects
WHERE object_type = 'event';
```

Events are defined by the Event Tracing for Windows model—this means that SQL Server Extended Events can be integrated with ETW. Like ETW events, Extended Events is categorized by:

- **Channel.** The event channel identifies the target audience for an event. These channels are common to all ETW events:
 - **Admin.** Events for administration and support.
 - **Operational.** Events for problem investigation.
 - **Analytic.** High-volume events used in performance investigation.
 - **Debug.** ETW developer debugging events.
 - **Keyword.** An application-specific categorization. In SQL Server, Extended Events event keywords map closely to the grouping of events in a SQL Trace definition.

A complete list of event keywords can be returned from **sys.dm_xe_map_values**.

Extended Events Event Keywords

```
SELECT map_value AS keyword
FROM sys.dm_xe_map_values
WHERE name = 'keyword_map'
ORDER BY keyword;
```

When you add, amend or remove an event from a package, you must refer to it with a two-part name: *packagename.eventname*.

For more information on **sys.dm_xe_objects**, see Microsoft Docs:

 **sys.dm_xe_objects (Transact-SQL)**

<http://aka.ms/bwkcmu>

Actions

Actions are responses to an event; you can use these responses to collect supplementary information about the context of an event at the time that it occurs. Each event may have a unique set of one or more actions associated with it. When an event occurs, any associated actions are raised synchronously.

 **Note:** Actions do not allow you to define responses to an event. Instead, actions are additional steps that occur within the Extended Events engine when an event is triggered. Most actions provide more data to be collected about an event.

SQL Server defines more than 50 different actions, which include:

- **Collect database ID**
- **Collect T-SQL stack**
- **Collect session ID**
- **Collect session's NT username**
- **Collect client hostname**

All available actions can be viewed in the DMV **sys.dm_xe_objects** under the **object_type** value action:

sys.dm_xe_objects; actions

```
SELECT * FROM sys.dm_xe_objects
WHERE object_type = 'action';
```

Predicates

Predicates are logical rules with which events can be selectively captured, based on criteria you specify. Predicates divide into two subcategories:

- Predicate comparisons. Comparison operators, such as “equal to”, “greater than”, and “less than”, which may make up a predicate filter. All predicate comparisons return a Boolean result (true or false).
- Predicate sources. Data items that may be used as inputs to predicate comparisons. These are similar to the column filters available when defining a SQL trace.

In addition to building logical rules, predicates are capable of storing data in a local context, which means that predicates based on counters can be constructed; for example, every n events or every n seconds.

Predicates are applied to an event using a WHERE clause—this functions like the WHERE clause in a Transact-SQL query.

All available predicates can be viewed in the DMV **sys.dm_xe_objects** under the **object_type** values **pred_source** and **pred_compare**.

sys.dm_xe_objects; predicates

```
SELECT * FROM sys.dm_xe_objects
WHERE object_type LIKE 'pred%'
ORDER BY object_type, name;
```

Targets and Sessions

Targets

Targets are the Extended Events objects that collect data. When an event is triggered, the associated data can be written to one or more targets. A target may be updated synchronously or asynchronously. The following targets are available for Extended Events:

- **Event counter.** The counter is incremented each time an event associated with a session occurs—synchronous.
- **Event file.** Event data is written to a file on disk—asynchronous.
- **Event pairing.** Tracks when events that normally occur in pairs (for example, lock acquired and lock released) do not have a matching pair—asynchronous.
- **Event Tracing for Windows.** Event data is written to an ETW log—synchronous.
- **Histogram.** A more complex counter that partitions counts by an event or action value—asynchronous.
- **Ring buffer.** A first-in, first-out (FIFO) in-memory buffer of a fixed size—asynchronous.

- Targets collect data from Extended Events sessions
- A session may write to multiple targets
- Targets may be synchronous or asynchronous
- An event will only be written once to a target
- A session links events to targets
 - Events may include actions
 - Events may be filtered with predicates
 - Sessions are isolated from one another
 - A session has a state (started or stopped)
 - A session has a buffer to hold event data as it is captured
- Use Watch Live Data to view data for a session without a target

The design of Extended Events is such that an event will only be written once to a target, even if multiple sessions are configured to send that event to the same target.

All available targets can be viewed in the DMV **sys.dm_xe_objects** under the **object_type** value **target**:

sys.dm_xe_objects; targets

```
SELECT * FROM sys.dm_xe_objects
WHERE object_type = 'target';
```

Sessions

A session links one or more events to one or more targets. You can configure each event in a session to include one or more actions, and to be filtered with one or more predicates. Once defined, a session can be started or stopped as required; it is possible to configure a session to start when the database engine starts.

A session may include events from more than one package. Sessions are isolated from one another; multiple sessions may use the same events and targets in different ways, without interfering with one another.

A session is configured with a buffer in which event data is held while a session is running, before it is dispatched to the session targets. The size of this buffer is configurable, as is a dispatch policy (how long data will be held in the buffer). You can also configure whether or not to permit data loss from the buffer if event data arrives faster than it can be processed and dispatched to the session target.

All active Extended Events sessions can be viewed in the DMV **sys.dm_xe_sessions**:

All active Extended Events sessions can be viewed in the DMV **sys.dm_xe_sessions**:

sys.dm_xe_sessions

```
SELECT * FROM sys.dm_xe_sessions;
```

For more information about active Extended Events DMVs, including **sys.dm_xe_sessions**, see Microsoft Docs:

 **Extended Events Dynamic Management Views**

<http://aka.ms/lmlj06>

For more information on the set of DMVs for accessing definitions for all Extended Events sessions, including **sys.server_event_sessions**, see Microsoft Docs:

 **Extended Events Catalog Views (Transact-SQL)**

<http://aka.ms/Cqon4y>

 **Note:** A session can be created without targets, in which case the session data is only visible using **Watch Live Data** in SSMS.

Types and Maps

Types and maps are metadata objects that make it easier to work with Extended Events data. Types and maps are not directly referenced in an Extended Events session definition.

Types

Internally, Extended Events data is held in binary. A type identifies how a binary value should be interpreted and presented when the data is queried.

All available types can be viewed in the DMV **sys.dm_xe_objects** under the **object_type** value **type**:

sys.dem_xe_objects; types

```
SELECT * FROM sys.dm_xe_objects
WHERE object_type = 'type';
```

- Types
 - Data type definitions for Extended Events data
- Maps
 - Lookup tables to convert integer values to text values

Maps

A map is a lookup table for integer values. Internally, many event and action data values are stored as integers; maps link these integer values to text values that are easier to interpret.

All available types can be viewed in the DMV **sys.dm_xe_map_values**:

sys.dem_xe_map_values

```
SELECT * FROM sys.dm_xe_map_values
ORDER BY name, map_key;
```

Demonstration: Creating an Extended Events Session

In this demonstration, you will learn how to:

- Create an Extended Events session.

Demonstration Steps

1. Start the **20762C-MIA-DC**, and **20762C-MIA-SQL** virtual machines, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Demofiles\Mod18** folder, run **Setup.cmd** as Administrator.
3. In the **User Account Control** dialog box, click **Yes** and wait for the script to finish.
4. Start **SQL Server Management Studio** and connect to the **MIA-SQL** database engine instance using Windows authentication.
5. On the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, navigate to the **D:\Demofiles\Mod18** folder, click **Demo.ssmssln**, and then click **Open**.
7. In Solution Explorer, double-click **Demo 1 - create xe session.sql**.

8. Select code under the comment that begins **Step 1**, and then click **Execute** to create an Extended Events session.
9. Select code under the comment that begins **Step 2**, and then click **Execute** to verify that the session metadata is visible.
10. Select code under the comment that begins **Step 3**, and then click **Execute** to start the session and execute some queries.
11. Select code under the comment that begins **Step 4**, and then click **Execute** to query the session data.
12. Select code under the comment that begins **Step 5**, and then click **Execute** to refine the session data query.
13. In Object Explorer, under **MIA-SQL**, expand **Management**, expand **Extended Events**, expand **Sessions**, expand **SqlStatementCompleted**, and then double-click **package0.ring_buffer**.
14. In the **Data** column, click the XML value, and note that this is the same data that is returned by the query under the comment that begins Step 4 (note that additional statements will have been captured because you ran the code earlier).
15. In Object Explorer, right-click **SqlStatementCompleted**, and then click **Watch Live Data**.
16. In the **Demo 1 - create xe sessions.sql** query pane, select the code under the comment that begins **Step 7**, and then click **Execute** to execute some SQL statements.
17. In the **MIA-SQL - SqlStatementCompleted: Live Data** pane. Wait for the events to be captured and displayed; this can take a few seconds. Other SQL statements from background processes might be captured by the session.
18. In the **Demo 1 - create xe sessions.sql** query pane, select the code under the comment that begins **Step 8**, and then click **Execute** to stop the session.
19. In Object Explorer, right-click **SqlStatementCompleted**, and then click **Properties**.
20. In the **Session Properties** dialog box, review the settings on the **General**, **Events**, **Data Storage**, and **Advanced** pages, if necessary referring back to the session definition under the comment that begins **Step 1**.
21. In the **Session Properties** dialog box, click **Cancel**.
22. In the **Demo 1 - create xe sessions.sql** query pane, select the code under the comment that begins **Step 10**, and then click **Execute** to drop the session.
23. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Question	
Which of the following statements about Extended Events is incorrect?	
Select the correct answer.	
	Extended Events can be viewed using Watch Live Data without starting a session.
	Extended Events sessions can be created using Transact-SQL commands.
	Extended Events is limited in what it can do and will soon be deprecated. Use SQL Trace whenever possible.
	Watch Live Data provides a real-time view of query execution statistics.

Lesson 2

Working with Extended Events

This lesson discusses using Extended Events. It covers common scenarios in which you might create Extended Events sessions for troubleshooting and performance optimization, in addition to the system_health Extended Events session, which captures several events relevant to performance tuning.

Lesson Objectives

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

- Configure Extended Events sessions.
- Configure Extended Events targets.
- Explain the system_health Extended Events session.
- Describe usage scenarios for Extended Events.
- Describe best practices for using Extended Events.

Configuring Sessions

As you have learned, Extended Events sessions are composed from several other object types, primarily events and targets. Sessions also have a number of configuration options that are set at session level:

- **MAX_MEMORY.** The amount of memory allocated to the session for use as event buffers, in kilobytes. The default value is 4 MB.
- **EVENT_RETENTION_MODE.** Specifies how the session will behave when the event buffers are full and further events occur:
 - ALLOW_SINGLE_EVENT_LOSS. An event can be dropped from the session if the buffers are full. A compromise between performance and data loss, this is the default value.
 - ALLOW_MULTIPLE_EVENT_LOSS. Full event buffers containing multiple events can be discarded. Minimal performance impact, but high data loss.
 - NO_EVENT_LOSS. Events are never discarded; tasks that trigger events must wait until event buffer space is available. Potential for severe performance impact, but no data loss.
- **MAX_DISPATCH_LATENCY.** The amount of time events will be held in event buffers before being dispatched to targets—defaults to 30 seconds. You may set this value to INFINITE, in which case the buffer is only dispatched when it is full, or the session is stopped.
- **MAX_EVENT_SIZE.** For single events larger than the size of the buffers specified by MAX_MEMORY, use this setting. If a value is specified (in kilobytes or megabytes), it must be greater than MAX_MEMORY.
- **MEMORY_PARTITION_MODE**
 - NONE. Memory is not partitioned. A single group of event buffers is created.
 - PER_NODE. A group of event buffers is created per NUMA node.

- Session configuration options:
 - MAX_MEMORY
 - EVENT_RETENTION_MODE
 - MAX_DISPATCH_LATENCY
 - MAX_EVENT_SIZE
 - MEMORY_PARTITION_MODE
 - STARTUP_STATE
 - TRACK_CAUSALITY

- PER_CPU. A group of event buffers is created per CPU.
- **STARTUP_STATE**. When set to ON, the session will start when SQL Server starts. The default value is OFF.
- **TRACK_CAUSALITY**. When set to ON, an identifier is added to each event identifying the task that triggered the event. With this, you can determine whether one event is caused by another.

For more information about configuring a session through Transact-SQL, see Microsoft Docs:

CREATE EVENT SESSION (Transact-SQL)

<http://aka.ms/b2eo2i>

Configuring Targets

Several Extended Events targets take configuration values when they are added to a session.

Event File

The event file target can be used to write session data to a file. It takes the following configuration parameters:

- **filename**. The file name to write to; this can be any valid file name. If a full path is not specified, the file will be created in the \\MSSQL\Log folder of the SQL Server instance on which the session is created.
- **max_file_size**. The largest size that the file may grow to; the default value is 1 GB.
- **max_rollover_files**. The number of files that have reached max_file_size to retain. The oldest file is deleted when this number of files is reached.
- **increment**. The file growth increment, in megabytes. The default value is twice the size of the session buffer.

- Targets collect data from Extended Events sessions
 - A session may write to multiple targets
 - Targets may be synchronous or asynchronous
 - An event will only be written once to a target
- A session links events to targets
 - Events may include actions
 - Events may be filtered with predicates
 - Sessions are isolated from one another
 - A session has a state (started or stopped)
 - A session has a buffer to hold event data as it is captured

For more information on configuring the event file target, see the SQL Server Technical Documentation:

Event File Target

<http://aka.ms/ixau4l>

Event Pairing

The event pairing target is used to match events that occur in pairs (for example, statement starting and statement completing, or lock acquired and lock released), and report on beginning events that have no matching end event. It takes the following configuration parameters:

- **begin_event**. The beginning event name of the pair.
- **end_event**. The end event name of the pair.
- **begin_matching_columns**. The beginning event columns to use to identify pairs.
- **end_matching_columns**. The ending event columns to use to identify pairs.
- **begin_matching_actions**. The beginning event actions to use to identify pairs.
- **end_matching_actions**. The ending event actions to use to identify pairs.

- **respond_to_memory_pressure.** Permit the target to discard events (and so reduce memory consumption) when memory is under pressure.
- **max_orphans.** The maximum number of unpaired events the target will collect. The default value is 10,000. When this number is reached, events in the target are discarded on a first-in, first-out basis.

For more information on configuring the event pairing target, see the SQL Server Technical Documentation:

Event Pairing Target

<http://aka.ms/lj7gng>

The system_health Extended Events Session

The system_health Extended Events session is created by default when a SQL Server 2008 or later version database engine instance is installed. The session is configured to start automatically when the database engine starts. The system_health session is configured to capture a range of events that are relevant for troubleshooting common SQL Server issues. These include:

- Details of deadlocks that are detected, including a deadlock graph.
- The **sql_text** and **session_id** when an error that has a severity of 20 (or higher) occurs.
- The **sql_text** and **session_id** for sessions that encounter a memory-related error.
- The **callstack**, **sql_text**, and **session_id** for sessions that have waited for more than 15 seconds on selected resources (including latches).
- The **callstack**, **sql_text**, and **session_id** for any sessions that have waited for 30 seconds or more for locks.
- The **callstack**, **sql_text**, and **session_id** for any sessions that have waited for a long time for preemptive waits. (A preemptive wait occurs when SQL Server is waiting for external API calls to complete; the trigger time varies by wait type.)
- The **callstack** and **session_id** for CLR allocation and virtual allocation failures (when insufficient memory is available).
- A record of any nonyielding scheduler problems.
- The **ring_buffer** events for the memory broker, scheduler monitor, memory node OOM, security, and connectivity. This tracks when an event is added to any of these ring buffers.
- System component results from **sp_server_diagnostics**.
- Instance health collected by **scheduler_monitor_system_health_ring_buffer_recorded**.
- Connectivity errors using **connectivity_ring_buffer_recorded**.
- Security errors using **security_error_ring_buffer_recorded**.

- The system_health Extended Events session is created by default
 - Starts when the SQL Server instance is started
 - Captures events useful for troubleshooting
 - Ring buffer and file targets

The system_health session writes data to two targets:

- A ring buffer target, configured to hold up to 5,000 events and to occupy no more than 4 MB.
- An event file target, composed of up to four files of 5 MB each.

 **Note:** The details of the system_health session are best understood by looking at its definition. You can generate a definition from SSMS:

1. Connect SSMS Object Explorer to a SQL Server instance on which you have administrative rights.
2. In the Object Explorer pane, expand **Management**, expand **Extended Events**, and then expand **Sessions**.

Right-click on the **system_health** node, click **Script As**, click **CREATE TO**, and then click **New Query Editor Window**. A script to recreate the system_health session will be generated.

Because both targets are configured to roll over and discard the oldest data they contain when they are full, the system_health session will only contain the most recent issues. On instances of SQL Server where the system_health session is capturing many events, the targets might roll over before you can examine specific events.

Usage Scenarios for Extended Events

Extended Events can be used to troubleshoot many common performance issues.

Execution Time-outs

When a Transact-SQL statement runs for longer than the client application's command time-out setting, a time-out error will be raised by the client application. Without detailed client application logging, it may be difficult to identify the statement causing a time-out.

This scenario is an ideal use case for the Extended Events event pairing target, using either of the following pairs:

- **sqlserver.sp_statement_starting** and **sqlserver.sp_statement_completed** (for systems using stored procedures for database access).
- **sqlserver.sql_statement_starting** and **sqlserver.sql_statement_completed** (for systems using ad hoc SQL for database access).

- Usage Scenario
- Execution time-outs
- Troubleshooting ASYNC NETWORK I/O issues
- Tracking error handling in T-SQL code
- Tracking recompliations
- tempdb latch contention
- Lock escalation
- Problematic page splits
- Troubleshooting orphaned transactions
- Tracking session waits
- Tracking database and object usage

When a time-out occurs, the starting event will have no corresponding completed event, and will be returned in the output of the event pairing target.

Troubleshooting ASYNC_NETWORK_IO

The ASYNC_NETWORK_IO wait type occurs when the database engine is waiting for a client application to consume a result set. This can occur because the client application processes a result set row-by-row as it is returned from the database server.

To troubleshoot this issue with Extended Events, capture the **sqlos.wait_info** event, filtering on **wait_type** value NETWORK_IO. The histogram target might be suitable for this investigation, using either the client application name or the client host name to define histogram groups.

Tracking Errors and Error Handling in Transact-SQL

Errors may be handled in Transact-SQL code by using TRY...CATCH blocks. Every error raises an event in Extended Events; this includes the errors handled by the TRY...CATCH blocks. You might want to capture all unhandled errors, or track the most commonly occurring errors, whether or not they are handled.

The **sqlserver.error_reported** event can be used to track errors as they are raised. The **is_intercepted** column can be used to identify if an error is handled in a TRY...CATCH block.

Tracking Recompilations

Query execution plan recompilations occur when a plan in the plan cache is discarded and recompiled. High numbers of plan recompilations might indicate a performance problem, and may cause CPU pressure. Windows performance counters can be used to track overall recompilation counts for a SQL Server instance, but more detail may be needed to investigate further.

In Extended Events, the **sqlserver.sql_statement_recompile** event can provide detailed information, including the cause of recompilation.

The histogram target can be used for tracking recompilations. You should group on **source_database_id** to identify the database with the highest number of recompilations in an instance; group on **statement/object_id** to find the most commonly recompiled statements.

tempdb Latch Contention

Latch contention in tempdb can occur due to contention for the allocation bitmap pages when large numbers of temporary objects are being created or deleted. This causes tempdb performance problems because all allocations in tempdb are slowed down.

The **latch_suspend_end** event tracks the end of latch waits by **database_id**, **file_id**, and **page_id**. With the predicate **divide_evenly_by_int64**, you can capture the contention specifically on allocation pages, because the different allocation bitmap pages occur at regular intervals in a database data file. Grouping the events using the histogram target should make it easier to identify whether latch waits are caused by contention for allocation bitmap pages.

Tracking Lock Escalation

Lock escalation occurs when more than 5,000 locks are required in a single session or under certain memory conditions.

The **sqlserver.lock_escalation** event provides the lock escalation information.

Tracking Problematic Page Splits

Page splits are of two types:

- Mid-page splits.
- Page splits for new allocations.

Mid-page splits create fragmentation and more transaction log records due to data movement.

Tracking page splits alone using the **sqlserver.page_split** event is inefficient because this does not differentiate the problematic mid-page splits and normal allocation splits. The **sqlserver.transaction_log** event can be used for tracking **LOP_DELETE_SPLIT** operations to identify the problematic page splits. A histogram target might be most suitable for this task, grouping either on **database_id** to find the database with the most page splits, or, within a single database, on **alloc_unit_id** to find the indexes with the most page splits.

Troubleshooting Orphaned Transactions

Orphaned transactions are open transactions where the transaction is neither committed nor rolled back. An orphaned transaction may hold locks and lead to more critical problems like log growth and blocking, potentially leading to a block on the whole SQL Server instance.

The **database_transaction_begin** and **database_transaction_end** events can be used with an event pairing target to identify the orphaned transactions. The **tsql_frame** action can be used to identify the line of code where the orphaned transaction started.

Tracking Session-Level Wait Stats

The wait stats available from the **sys.dm_os_wait_stats** DMV are aggregated at instance level, so it's not a fine-grained troubleshooting tool. Although you can track wait stats by session with the additional **sys.dm_exec_session_wait_stats** DMV, this may not be suitable for use in a busy system with many concurrent database sessions.

The **sqlos.wait_info** event can be used to track waits across multiple concurrent sessions.

Tracking Database and Object Usage

Tracking database and objects usage helps to identify the most frequently used database and most frequently used objects within a database. You might use this information to guide your optimization efforts, or to prioritize objects for migration to faster storage or memory-optimized tables.

The **sqlserver.lock_acquired** event can help with tracking the usage in most cases. For database usage, a histogram can target grouping on **database_id**. Object usage can be tracked by tracking SCH_M or SCH_S locks at the object resource level by grouping on **object_id** in a histogram target.

Extended Events Best Practices

Run Extended Events Sessions Only When Needed

Although Extended Events is a lightweight logging framework, each active session has an overhead of CPU and memory resources. You should get into the practice of only running Extended Events sessions you have created when you have to troubleshoot specific issues.

- Run Extended Events sessions only when you need them
- Use the SSMS GUI to browse available events
- Understand the limitations of the ring buffer target
- Consider the performance impact of collecting query execution plans
- Deadlock graph format
 - `xml_deadlock_report`
 - `database_xml_deadlock_report`

Use the SSMS GUI to Browse Available Events

The Events page of the Extended Events GUI in SSMS brings all the metadata about individual events together into one view; this view makes understanding the information that Extended Events makes available to you easier than querying the DMVs directly.

Understand the Limitations of the Ring Buffer Target

When using a ring buffer target, be aware that you might not always be able to view all the events contained in the ring buffer. This is due to a limitation of the **sys.dm_xe_session_targets** DMV; the DMV is restricted to displaying 4 MB of formatted XML data. Because Extended Events data is stored internally as unformatted binary, it is possible that the data in a ring buffer will, when converted to formatted XML, exceed the 4 MB limit of **sys.dm_xe_session_targets**.

You can test for this effect by comparing the number of events returned from a ring buffer in XML with the count of events returned in the XML document header—or check the value of the truncated attribute in the XML header.

A query comparing values for the system_health session:

Ring Buffer: Number of Events in XML Compared with Header

```
SELECT x.target_data.value('(RingBufferTarget/@eventCount)[1]', 'int') AS event_count,
       x.target_data.value('count(RingBufferTarget/event)', 'int') AS node_count,
       x.target_data.value('(RingBufferTarget/@truncated)[1]', 'bit') AS output_truncated
  FROM (
    SELECT CAST(target_data AS xm1) AS target_data
      FROM sys.dm_xe_sessions AS xs
     JOIN sys.dm_xe_session_targets AS xst
       ON xs.address = xst.event_session_address
      WHERE xs.name = N'system_health'
      AND xst.target_name = N'ring_buffer'
  ) AS x
```

To avoid this effect, you can:

- Use a file-based target (event file target or ETW target).
- Reduce the size of the MAX_MEMORY setting for the ring buffer to reduce the likelihood that the formatted data will exceed 4 MB. No single value is guaranteed to work; you may have to try a setting and be prepared to adjust it to minimize the truncation effect while still collecting a useful volume of data in the ring buffer.

 **Note:** This effect is not strictly limited to the ring buffer target; it can occur on any target that stores output in memory buffers (ring buffer target, histogram target, event pairing target, and event counter target). However, it is most likely to affect the ring buffer target because it stores unaggregated raw data. All the other targets using memory buffers contain aggregated data, and are therefore less likely to exceed 4 MB when formatted.

Performance Impact of Collecting Query Execution Plans

Three events can be used to collect query execution plans as part of an Extended Events session:

- **query_post_compilation_showplan.** Returns the estimated query execution plan when a query is compiled.
- **query_pre_execution_showplan.** Returns the estimated query execution plan when a query is executed.
- **query_post_execution_showplan.** Returns the actual query execution plan when a query is executed.

When using any of these events, you should consider that adding them to a session, even when predicates are used to limit the events captured, can have a significant impact on the performance of the database engine instance. This effect is most marked with the **query_post_execution_showplan** event. You should limit your use of these events to troubleshooting specific issues; they should not be included in an Extended Events session that is always running.

Deadlock Graph Format

Deadlock graphs collected by the **xml_deadlock_report** and **database_xml_deadlock_report** events are in a different format from the deadlock graphs produced by SQL Server Profiler; with these, you can use deadlock graphs captured by Extended Events to represent complex deadlock scenarios involving more than two processes. If saved as an .xdl file, both formats of deadlock graph can be opened by SSMS.

Demonstration: Tracking Session-Level Waits

In this demonstration, you will see how to use Extended Events to report on wait types by session.

Demonstration Steps

1. In SSMS, in Solution Explorer, double-click **Demo 2 - track waits by session.sql**.
2. In Object Explorer, expand **Management**, expand **Extended Events**, right-click **Sessions**, and then click **New Session**.
3. In the **New Session** dialog box, on the **General** page, in the **Session name** box, type **Waits by Session**.
4. On the **Events** page, in the **Event library** box, type **wait**, and then, in the list below, double-click **wait_info** to add it to the **Selected events** list.
5. Click **Configure** to display the **Event configuration options** list.
6. In the **Event configuration options** list, on the **Global Fields (Actions)** tab, select the **session_id** check box.
7. On the **Filter (Predicate)** tab, click **Click here to add a clause**.
8. In the **Field** list, click **sqlserver.session_id**, in the **Operator** list, click **>**, and then in the **Value** box, type **50**. This filter will exclude most system sessions from the session.
9. On the **Data Storage** page, click **Click here to add a target**.
10. In the **Type** list, click **event_file**, in the **File name on server** box, type **D:\Demofiles\Mod18\waitbysession**, in the first **Maximum file** size box, type **5**, in the second **Maximum file** size box, click **MB**, and then click **OK**.
11. In Object Explorer, under **Sessions**, right-click **Waits by Session**, and then click **Start Session**.
12. In File Explorer, in the **D:\Demofiles\Mod18** folder, right-click **start_load_1.ps1**, and then click **Run with PowerShell**. If a message is displayed asking you to confirm a change in execution policy, type **Y**, and then press Enter. Leave the workload to run for a minute or so before proceeding.
13. In SSMS, in the **Demo 2 - track waits by session.sql** query pane, select the code under the comment that begins **Step 14**, click **Execute**, and then review the results.
14. Select the code under the comment that begins **Step 15**, and then click **Execute** to stop and drop the session, and to stop the workload.
15. In File Explorer, in the **D:\Demofiles\Mod18** folder, note that one (or more) files with a name matching **waitbysession*.xel** have been created.
16. Close File Explorer, and then close Windows PowerShell®.
17. Keep SQL Server Management Studio open for the next demonstration.

Categorize Activity

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

Items	
1	Ring buffer target
2	Event file target
3	Histogram target
4	Event tracking for Windows target
5	Event pairing target
6	Event counter target

Category 1	Category 2
Written to Memory Buffers	Written to File on Disk

Lesson 3

Live Query Statistics

Live Query Statistics gives immediate insight into the execution of a query, making the debugging of long-running queries easier to resolve. You can see each step of the plan exactly as it executes.

Lesson Objectives

After completing this lesson, you will be able to:

- Enable Live Query Statistics for one or all sessions.
- Use Live Query Statistics to view execution plans for queries as they are running.

What Is Live Query Statistics?

Developers and DBAs now have the ability to view a live execution plan for a query while the query is running. This is helpful when debugging code, as you can see the flow of operations while the query executes.

Live Query Statistics will work with SQL Server 2014 and later. Some performance degradation is likely to occur as a result of running Live Query Statistics, so it should not be used in a production environment, unless a query is causing problems.

Live query statistics is a feature of SSMS, which is now available independently of SQL Server. You can download the latest version of SSMS free of charge here:

 **Download SQL Server Management Studio (SSMS)**

<http://aka.ms/o4vgkz>

- Compatible with SQL Server 2014 and later
- View a query plan live as the query executes
- Gain insight into long-running queries
- Affects query performance and may reduce speed
- Live Query Statistics is a feature of SSMS

Enabling Live Query Statistics

You can enable Live Query Statistics for the current session only, returning the live plan for your current query, or for all sessions.

There are two ways to enable Live Query Statistics for the current session:

- On the **Query** menu, click **Include Live Query Statistics** while in the query window.
- Right-click in the query window, and then click **Include Live Query Statistics**.

- Enable for the current session and view statistics in the results pane
- Enable for all sessions and view with Activity Monitor

When the query executes, you will see a new tab in the results pane.

Enabling Live Query Statistics for all sessions adds the statistics to Activity Monitor. There are two ways that you can do this:

- Execute **SET STATISTICS PROFILE ON** or **SET STATISTICS XML ON** in the target session. Both return the same data, but **SET STATISTICS XML ON** returns the data as XML, enabling another application to process the results.
- Enable the **query_post_execution_showplan** extended event. This is a server setting that will affect all settings.

To learn more about using Extended Events, see MSDN:

 **Monitor System Activity Using Extended Events**

<http://aka.ms/n8n81>

 **Note:** When enabling Live Query Statistics for all sessions, remember that there is a performance overhead to running it.

Demonstration: Enable Live Query Statistics for a Session

In this demonstration, you will see how to:

- Enable Live Query Statistics.
- View the query execution plan.
- Disable Live Query Statistics.

Demonstration Steps

1. In SSMS, in Solution Explorer, double-click **Demo 3 - live query statistics.sql** script file.
2. Highlight the script under the **Step 1** description, and then click **Execute**.
3. Highlight the script under the **Step 2** description, and then click **Execute**.
4. On the **Query** menu, click **Include Live Query Statistics**.
5. Highlight the script under the **Step 4** description, and then click **Execute**.
6. On the **Query** menu, click **Include Live Query Statistics**.
7. Close SQL Server Management Studio, without saving any changes.

Check Your Knowledge

Question
Which of the following statements will enable Live Query Statistics for all sessions in Activity Monitor?
Select the correct answer.
<input type="checkbox"/> ALTER DATABASE SET STATISTICS ON
<input type="checkbox"/> SET LIVE QUERY STATISTICS ON
<input type="checkbox"/> ENABLE LIVE QUERY STATISTICS
<input type="checkbox"/> SET STATISTICS PROFILE OFF
<input type="checkbox"/> SET STATISTICS XML ON

Lesson 4

Optimize Database File Configuration

This lesson looks at the options for configuring database files, including tempdb.

Lesson Objectives

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

- Explain the importance of tempdb.
- Configure tempdb.
- Look at options for storing data and log files.

Improving Performance with tempdb

tempdb is the temporary database shared by all the databases on a SQL Server instance. It plays a critical role in ensuring your databases perform well. When SQL Server is heavily loaded, tempdb can become a bottleneck, unless it is properly configured. To understand why, it is helpful to consider how it is used.

Understanding tempdb

- tempdb is recreated each time SQL Server starts, and is dropped when SQL Server is stopped. Like all databases, it inherits from the model database.
- tempdb is a shared temporary database for all databases on a SQL Server instance—this means that a single database can cause performance problems for all the databases on that instance.
- tempdb is a system database. It is not a database you normally work with directly, but a temporary storage area for objects created by users, internal objects, and version stores. These objects are created and destroyed during normal operation. User objects include local and global temporary tables, table variables, temporary indexes, and tables from table-valued functions.
- tempdb is used to hold intermediate results by queries that include GROUP BY, ORDER BY, or UNION clauses where intermediate results need to be stored when they can't be cached.
- tempdb holds work tables for hash join and hash aggregate operations, in addition to work tables for cursor, spool operations, and large object (LOB) storage.

In addition, tempdb acts as a version store for online index operations, and transactions with READ COMMITTED SNAPSHOT and SNAPSHOT isolation levels.

All the databases on an instance rely on tempdb for fast reads and writes, and most SQL Server databases generate a great deal of tempdb activity. If tempdb runs out of space, errors will occur causing a significant amount of disruption in a production environment.

- Tempdb is:
 - Shared between all databases on a SQL Server instance
 - A temporary database; recreated at startup
 - Used for a variety of tasks such as holding user created objects, work area for query intermediate results, and work tables
- For best performance:
 - Store separately from data or log files
 - Increase the number of tempdb data files to reduce latch contention
 - Store on fast storage media

Also, if there are latch contention issues, performance will suffer. A latch is a type of lock made on GAMs (Global Allocation Maps), SGAMs (Shared Global Allocation Maps) and PFS (Page Free Space) pages. These pages are used to determine where tempdb data can be written. If tempdb has only one data file, it is easy to run into latch contention issues. With several data files, the load is spread between many PFS, GAM, and SGAM pages to determine where data can be written.

Improving Performance with tempdb

There are a number of things you can do to ensure that tempdb does not become a bottleneck:

- Store tempdb on separate storage. You get the best performance when database data files, log files, and tempdb files are held separately.
- Store tempdb on fast storage. Use Solid State Disks (SSDs) to improve performance of tempdb data.
- Increase the size of tempdb.
- Enable autogrowth for tempdb.
- Use multiple data files for tempdb.
- Enable Database Instant File Initialization to speed up file initialization.



Best Practice: Use fast storage for tempdb. Unlike a log file that writes records to disk sequentially, tempdb has varied reads and writes, and benefits from fast storage media.

SQL Server Installation

At installation the default settings for tempdb depend on the number of processors installed on your server. This means that performance should be improved for many SQL Server instances without needing to make any changes after installation.

There are different aspects to configuring tempdb. A full discussion of how to configure tempdb is an involved topic. If you want to find out more, see TechNet's white paper:



Working with TempDb in SQL Server 2005

<http://aka.ms/dzi7g8>

Configuring tempdb

You can configure both tempdb data files, and log files. The options include:

- **Number of files.** The default value is either eight or the number of logical processor cores on your server. Microsoft recommends that, if you have less than eight processors, then the number of tempdb data files should be the same as the number of processors. If your server has more than eight processors, and you experience contention issues, increase the number of data files in multiples of four; or make changes to the way your databases work to reduce tempdb contention. For optimal performance, you should only have the necessary number of tempdb data files.

- Number of tempdb files = number of processors
 - Up to eight, then add in increments of four
- 8 MB default initial file size
 - All files must be the same size
 - Ensure that the initial file size is sufficient for normal usage
- Autogrowth default: 64 MB
- Data directories
 - Local storage
 - Shared storage
 - SMB file shares

- **Initial size (MB).** Make all tempdb data files the same size. This is important because of the algorithm used for page allocation. If files differ in size, you will not get the intended performance improvements. The page size required will depend on the database workload—8 MB is the default for primary and secondary tempdb data files.
- **Total initial size (MB).** The total size of all tempdb data files.
- **Autogrowth (MB).** The number of megabytes that each data file will increase by when they run out of space. All tempdb data files will grow by the same amount, and at the same time. The recommended, and default size, is 64 MB. You should always allow tempdb data files to grow—if they run out of space, you will get errors instead of the intended operations.
- **Total autogrowth (MB).** The total number of megabytes when all tempdb data files autogrow.
- **Data directories.** You can store your tempdb data files in different directories. SQL Server supports local disk storage, shared storage, and SMB file shares.
- **Log file initial size (MB).** The log is used to roll back transactions. However, tempdb cannot be backed up, and uses minimal logging—there is no point-in-time recovery with tempdb.
- **Log file autogrowth (MB).** The default value is 64 MB. The maximum size is 2 TB. Always allow the tempdb log file to grow.
- **Log directory.** Path for the tempdb log file. There can only be one tempdb log file, so there is only one directory path.

 **Best Practice:** Ensure tempdb files are large enough for normal workload without relying on autogrowth. For best performance, autogrowth should handle the unexpected, not the everyday.

Performance Improvements

SQL Server performance is improved by tempdb configuration in several ways:

- Allocations in tempdb all use uniform extents. This reduces blocking issues when SQL Server is heavily loaded, and means that trace flag 1118 is not needed.
- Multiple data files all grow by the same amount, at the same time. If data files are different sizes, you don't get the same benefit in reducing contention.
- The primary filegroup has the AUTOGROW_ALL_FILES property set to ON, and this cannot be changed.

 **Best Practice:** When configuring a SQL Server instance for failover clustering, ensure the directories are valid for all cluster nodes. If the tempdb directory(s)—including log directory—are not on the failover target node, SQL Server will not fail over.

For more information about tempdb settings, see Microsoft Docs:

 **tempdb Database**

<https://aka.ms/Yntqsx>

Optimize Database File Configuration

SQL Server uses both data files and log files. Each transaction is written to both a data file, and a log file, to ensure that the database is never in an inconsistent state. In the event of a system failure, SQL Server can go to the log file to get the latest transactions.

Storing data files and log files on separate storage media improves performance.

To help you understand read and write activity on your database, use system DMVs to understand whether you need to store your data files and log files separately. For example, the

`sys.dm_io_virtual_file_stats` DMV gives you I/O statistics for data and log files. Use the results in combination with baseline performance statistics.

Provide a valid database ID and file ID to get a variety of I/O statistics:

sys.dm_io_virtual_file_stats

```
SELECT *
FROM sys.dm_io_virtual_file_stats(database_id, file_id)
```

- Consider splitting data files and log files onto separate storage media
- Use DMVs to understand I/O loading
- Sys.`dm_io_virtual_file_stats`
- Use in combination with baseline statistics

Check Your Knowledge

Question	
Which of the following statements is correct?	
Select the correct answer.	
	Each SQL Server user database has one tempdb.
	tempdb is a shared resource between all databases on a SQL Server instance.
	The number of tempdb databases depends on the number of logical processors on the server.
	You can have any number of tempdb databases on an instance, but the exact number is hidden.
	tempdb should be deleted when not in use.

Lesson 5

Metrics

This lesson looks at gathering SQL Server metrics, including creating a baseline, using dynamic management objects (DMOs) and Windows Performance Monitor.

Lesson Objectives

After completing this lesson, you will be able to:

- Monitor operating system and SQL Server performance metrics.
- Compare baseline metrics to observed metrics.
- Use PerfMon.
- Use Dynamic Management Objects to get performance metrics.

Generating Baseline Metrics

Baselines

A baseline is the quantified normal or usual state of a SQL Server environment. A SQL Server baseline tells you how a system typically performs. Benefits of having a baseline include:

- You can compare baselines with the current system state to make comparisons.
- You can identify unusual activity, and proactively tune or resolve problems.
- Making capacity planning easier.
- Making troubleshooting easier.

- Baseline metrics provide information about the normal state of a SQL Server instance
- Consider capturing
 - Performance Monitor counters
 - Dynamic Management View (DMV) output
 - Trace data
- Consider data capture frequency by type of data

To create a baseline, capture SQL Server diagnostic data over time and calculate the average. For example, resource usage may be high during business peak hours, and lower during off-peak hours. Alternatively, it might be high during weekends when weekly reports are generated. Multiple baselines may be needed for different situations.

Benchmarks

A benchmark is a standard point of reference against which subsequent metrics can be compared. A benchmark is different from a baseline. For example, the benchmark for a stored procedure execution time might be two seconds. The baseline will give the usual or normal execution time for stored procedures. If the execution time for the specific stored procedure is less than the baseline, then it is considered good; otherwise, the stored procedure needs to be optimized.

What to Capture?

Many data points are available to be collected. Consider how you will use the data before capturing a specific data point—how can it help you diagnose and foresee performance problems? To start with, consider collecting the following:

- **System usage.** System usage is described in terms of CPU, I/O, and memory consumption. You can check these basic Performance Monitor counters against the current values, in case of sudden performance degradation. These values can also be used to define a system usage trend, which will help further with capacity planning.
- **SQL Server configurations.** These are instance-level or database-level configuration settings, such as max server memory, degree of parallelism, or auto shrink.
- **Database size information.** The system will stop when it runs out of storage space. Capture database and file size information to help you react proactively to space issues, thereby preventing system downtime.
- **Wait statistics.** Wait statistics are helpful when troubleshooting SQL Server performance issues. They give insight into the root cause of an issue and are helpful when optimizing a slow system.

Data Capture Frequency

After you decide what to capture, the next step is to decide the frequency of the data capture. The frequency governs the amount of data that is captured. Too much data will result in high storage costs; too little data will not provide sufficient understanding of the system. The frequency also depends on the type of data—for example, Performance Monitor data can be captured every 15 seconds or so. However, SQL instance configuration data would be recorded much less often.

It is also necessary to have a data retention strategy. It is not advisable to keep a year or six months' worth of baseline data, because this data is not likely to be useful. In addition, capturing data during off-peak hours will only add additional data to the storage without being useful.



Best Practice: Capture relevant data and keep it only for as long as it is useful—typically three to four months.

Collecting Data Using DMVs

DMVs are views onto the internal workings of SQL Server. As the name implies, they are dynamic in that they provide internal metadata, including internal memory structures, as queries are executed. The data that DMVs expose will help you to optimize the performance of your SQL Server instance. You can capture data from DMVs by using Transact-SQL queries, which can be saved and used as required.

DMVs are useful in capturing a baseline because data such as wait statistics, SQLoS information, and cached query plans, cannot be obtained through other sources. Data collection through certain DMVs may incur overhead on the SQL Server instance. For example, obtaining index fragmentation details using the `sys.dm_db_index_physical_stats` DMV for all indexes in a large database might take time to return and can negatively affect SQL Server performance. Index DMVs, which were discussed in Module 6, should be monitored regularly.

- Collect data using DMOs
- Data that should be collected regularly includes:
 - Indexes that are used regularly, and missing indexes
 - User activity
 - Tasks that are waiting for something
 - Missing statistics

In addition, other important DMVs that you should consider monitoring are:

Users Sessions

SQL Server uses session_ids to track users connected to SQL Server. The sys.dm_exec_session DMV collects information about sessions including time connected, CPU usage, memory usage, and more.

For example, to display a list of users connected to SQL Server, plus the time they have been connected in milliseconds:

sys.dm_exec_session

```
SELECT login_name, SUM(total_elapsed_time)/1000 AS time_connected_seconds
    FROM sys.dm_exec_sessions
    GROUP BY login_name;
```

SQL Server Waits

- **sys.dm_os_waiting_tasks**: displays every task waiting for something.
- **sys.dm_os_wait_stats**: displays an aggregation of all wait times since the SQL Server instance was started.

Query Performance

Statistics play an important role in ensuring queries are executed in the most efficient way. Use the following DMOs in combination to identify queries that have missing statistics:

- **sys.dm_exec_cached_plans**: returns plan handles for cached plans.
- **sys.dm_exec_sql_text**: returns the Transact-SQL text for a given plan handle.
- **sys.dm_exec_query_plan** returns the plan for the specified plan handle.

Performance Monitor

Windows Performance Monitor is a snap-in for Microsoft Management Console (MMC) that comes with the Windows Server operating system. Performance Monitor provides graphical views onto the performance of the operating system and the server. Performance Monitor is helpful because it includes data about the server and its operating system, in addition to SQL Server. You can view performance data either from log files or in real time.

- Windows Performance Monitor is a snap-in for Microsoft Management Console
- Displays real-time performance data
- Sales performance data to text files or a database
- Enables creation of custom data collector set
- Can respond to alerts
- Start by typing **Performance Monitor** from the start screen

What Does Performance Monitor Do?

You can use Performance Monitor to:

- Display real-time system performance data in three formats—line graph, histogram, and report.
- Monitor overall system health, SQL Server health, or the health of other applications by selecting counters from the available performance objects.
- Record current performance counter values in text files and databases to analyze later. You can analyze the performance data by using file manipulation techniques or Transact-SQL queries against the appropriate database.

- Create custom sets of performance counters known as a “data collector set” that you can then schedule to run as appropriate.
- Configure and respond to alerts. For example, when a specified threshold is reached, start a particular data collector set or a particular program.

You can use Performance Monitor for real-time monitoring, and to establish a baseline for SQL Server performance. You can collect performance data over time and analyze it to calculate workload characteristics such as peak and off-peak hours, average CPU usage, memory usage, and more. It is often useful to demonstrate performance gains or losses following a system change. Use Performance Monitor to compare counter values before and after implementation.

Performance Monitor is a lightweight tool, with minimal performance overhead for sampling intervals of greater than one second. The default or optimal sampling interval is 15 seconds. The amount of I/O that Performance Monitor generates will depend on the number of counters, the sampling interval, and the underlying storage. If there is an I/O issue, consider saving the performance data on separate storage and only enable the counters that you need.

Starting Performance Monitor

1. To start Performance Monitor, type **Performance Monitor** from the start screen. The Performance Monitor screen appears with a navigation tree in the left pane. Alternatively, type **perfmon** from the Run command window.
2. There are three views:
 - a. Monitoring Tools
 - b. Data Collector Sets
 - c. Reports
3. The right pane displays associated data.

Performance Monitor Counters

Performance Monitor enables you to create a baseline, or compare the current system performance against an established baseline, by using the captured data.

Getting Started with Performance Monitor

You cannot use the real-time performance data for historical analysis. It can only be used to monitor current system state and compare against the established baseline. To monitor real-time performance using Performance Monitor, follow these steps:

1. Start Performance Monitor.
2. In the leftmost pane, expand **Monitoring Tools**, and then click **Performance Monitor**. This will open the Performance Monitor window in the rightmost pane.
3. To add the counters to monitor, click the **Plus Sign**.
4. When you have added the required counters, click **OK** to view performance information.

- Performance Monitor allows you to:
 - Monitor real-time system performance
 - Collect data in response to events
 - Collect scheduled data
- Performance counts include:
 - CPU usage
 - Memory usage
 - Disk usage
 - SQL Server statistics

By default, Performance Monitor shows the last 100 seconds of data. This value can be changed from the Performance Monitor properties window that is opened by right-clicking on Performance Monitor and selecting properties.

There are three different types of graph available in Performance Monitor: Line, Histogram, and Report.

Performance Monitor provides data collector sets to automate collection of selected counters. There are two types of data collector sets: system and user defined. The system set includes OS and network specific counters but does not include SQL Server counters. The data collector sets can also be triggered to start when a specific event or alert occurs. For example, a data collector set can be started when the available memory is less than 100 MB. To set up and schedule a data collector set to collect performance counters, follow these steps:

1. Start Performance Monitor.
2. In the Performance Monitor window, in the leftmost pane, expand **Data Collector Sets**, right-click **User Defined**, click **New**, and then click **Data Collector Set**. The **Create new Data Collector Set** dialog box will appear.
3. In the **Create new Data Collector Set** dialog box, type a name, click **Create manually**, and then click **Next**.
4. In the **Create data logs** section, select **Performance counter**, and then click **Next**.
5. Choose the appropriate performance counters.
6. Specify a sampling interval, and then click **Finish**. You can now schedule the data collector or execute manually as required.

To configure the data collector set:

1. Right-click the data collector set you created, and then click **Properties**.
2. On the **Schedule** tab, click **Add**, and in the **Folder Action** dialog box, schedule the collection.
3. On the **Stop Condition** tab, specify a duration or maximum size for the set.

The data collector set will use your schedule to start collecting data or you can start it manually by using the **Action** menu.

Important Performance Counters

This section describes some of the important performance counters you could collect.

CPU Usage

- Processor:
 - %Processor Time
 - %Privileged Time
- Process (sqlservr.exe):
 - %Processor Time
 - %Privileged Time

The %Processor Time counter gives information about the total CPU usage and should be monitored for each available CPU. The Process (sqlservr.exe)/% Processor Time counter details how much CPU the SQL Server instance is using. If high CPU usage is a result of another application, you should investigate options for tuning that application. Occasional CPU spikes may occur and should not cause concern, but you should investigate prolonged values of greater than 80 percent.

Memory Usage

- Memory
 - Available Mbytes
- SQL Server: Buffer Manager
 - Lazy writes/sec
 - Buffer cache hit ratio
 - Page life expectancy
 - Page reads/sec
 - Page writes/sec
- SQL Server: Memory Manager
 - Total Server Memory (KB)
 - Target Server Memory (KB)

The Memory/Available Mbytes counter shows the amount of physical memory that is immediately available for allocation to a process or for system use. Ideally, this should be more than 300 MB. When it drops to less than 64 MB, on most servers, Windows will display low memory notifications. The SQLOS reacts to these notifications by reducing its memory usage.

The page life expectancy counter shows the amount of time that data pages stay in the buffer pool. Ideally, this should be more than 300 seconds. If page life expectancy is less than 300 seconds, investigate other buffer manager counters to get to the root cause. If the Lazy writes/sec counter is consistently non-zero, along with low page life expectancy, and high page reads/sec and page writes/sec counters, there is a buffer pool contention. The buffer cache hit ratio counter shows how often SQL Server gets a page from the buffer rather than the disk. This should ideally be close to 100 percent.

The total server memory is the current amount of memory that an instance of SQL Server is using. The target server memory is the amount of memory that is allocated to a SQL Server instance. Ideally, total server memory is equal to target server memory on a stable system. If total server memory is less than the target server memory, it means that SQL Server is still populating the cache and loading the data pages into memory. A sudden decrease in total server memory indicates a problem and needs further investigation.

Disk Usage

- Physical Disk
 - Avg. Disk sec/Read
 - Avg. Disk Bytes/Read
 - Avg. Disk sec/Write
 - Avg. Disk Bytes/Write
- Paging File
 - %Usage
- SQL Server: Access Methods
 - Forwarded Records/sec
 - Full Scans/sec
 - Index Searches/sec

- Page splits/sec

The Avg. Disk sec/Read counter shows the average time taken, in seconds, to read data from disk. Similarly, the Avg. Disk sec/Write counter shows the average time taken, in seconds, to write data to disk. High values of these counters may not indicate hardware issues. Poorly tuned queries and missing or unused indexes may result in high I/O usage.

The Forwarded Records/sec value should be less than 10 per 100 batch requests/sec. Consider creating a clustered index if this counter is consistently high. A high value for the Full Scans/sec counter may cause high CPU usage. Full scans on smaller tables are acceptable; however, a large number of scans on big tables should be investigated. The Counter Page splits/sec value should ideally be less than 20 per 100 batch requests/sec. A high number of page splits may result in blocking, high I/O, or memory pressure. Set an appropriate fill factor value to balance out page splits.

SQL Server Statistics

- SQL statistics
- Batch requests/sec
- SQL compilations/sec
- SQL recompilations/sec
- SQL Server: General Statistics
- User connections
- Logins/sec
- Logouts/sec

The batch requests/sec value is the number of Transact-SQL batch requests that SQL Server receives per second. The SQL compilations/sec counter shows the number of times per second that SQL compilations have occurred. A high number of SQL compilations and recompilations may cause a CPU bottleneck. The SQL compilations/sec value should ideally be less than 10 percent of the number of batch requests/sec; SQL recompilations should ideally be less than 10 percent of the total number of SQL compilations/sec.

The user connections counter shows the number of users who are currently connected to a SQL Server instance. The logins/sec and logouts/sec values should ideally be less than two. If the value is consistently greater than two, it means that the connection pooling is not being correctly used by the application.

This is not an exhaustive list of counters to be considered, but these are good starting points when baselining SQL Server.

Analyzing Performance Monitor Data

The data you collect will not be useful until you can analyze it to get meaningful information. You can analyze Performance Monitor data by using Microsoft Excel® or by importing the performance logs into a database.

Microsoft Excel

The performance data, in csv format, can be analyzed manually in Excel. If the data is collected in binary format, the binary log file can be converted to csv format with the relog command-line utility. The relog utility ships with Windows and does not require a separate installation.

The following example shows a typical command to convert a binary log file to csv format by using the relog utility:

Using Relog to Convert to csv

```
relog <binary file path> -f csv -o <csv file path>
```

Analyzing data in Excel can be time-consuming. The column headers need to be formatted, the performance data requires formatting, and then aggregate columns need to be added to get the maximum and minimum standard deviation for the counter values. This becomes even more tedious when more than one file is to be analyzed.

Database

The performance data can be imported into SQL Server and analyzed by using Transact-SQL statements. The performance data is imported into a database manually, loaded by using SQL Server Integration Services, or loaded by using the relog utility. The simplest method is probably the use of the relog command-line utility.

The following example shows the syntax to import a performance log into a database by using the relog utility:

Using Relog to Import Into a Database

```
relog <binary file path> -f SQL -o SQL:<ODBC Connection>!<display string>
```

The relog utility accepts a binary log file and inserts it into the database that the Open Database Connectivity (ODBC) connection specifies. The display string identifies the binary log file or the data collector set within the database. The relog utility imports data into three different tables, as follows:

- **DisplayToID.** This lists each data collector set that is imported into the database. A unique identifier uniquely identifies each data collector set. The data collector is identified by the display string value that is specified when importing the data, as shown in the preceding relog command syntax. The table also contains the number of records that are imported and the log start and log stop time.
- **CounterDetails.** This contains one row for each counter that is present in the performance log file. Every counter is uniquely identified by a unique counter id. Each counter has an associated machine name. This is helpful in identifying counter values from different computers.

- To analyze performance monitor data you can use:
 - Microsoft Excel
 - Use relog to create a csv file
 - Apply aggregations in a worksheet
 - A database
 - Use relog to import the data
 - Analyze the data using Transact-SQL

- **CounterData.** The CounterData table stores the actual counter values. The important columns are GUID, counterID, counter value, and counterdatetime. The GUID columns link to the DisplayToID GUID column. The counterID columns link to the CounterDetails counterID column. The counter value column contains the actual counter value, and the counterdatetime column contains the time that the value was recorded.

These three tables can be queried to get different counter values, as shown in the following example:

Counter Values

```
SELECT dd.DisplayString, cd.CounterDateTime,
       cdt.ObjectName, cdt.CounterName,
       cd.CounterValue
  FROM dbo.CounterData cd
    JOIN dbo.DisplayToID dd ON cd.GUID=dd.GUID
    JOIN dbo.CounterDetails cdt ON cd.CounterID=cdt.CounterID

 WHERE did.DisplayString='SQLPerf'
 ORDER BY cdt.ObjectName, cdt.CounterName, cd.RecordIndex;
```

The preceding query will list the counter values for the display string "SQLPerf." To get the data from all display strings, remove the filter on the **DisplayString** column in the preceding query.

You can aggregate data to form a baseline, and you can import data from multiple data collector sets with different display strings, and then compare them to diagnose issues.

A sample query to get aggregate data from a particular display string or data collector set:

Aggregating Data for a Single Display String or Data Collector Set

```
SELECT CONVERT(VARCHAR(10), cd.CounterDateTime, 101) AS CounterDateTime,
       RTRIM(cdt.ObjectName) AS ObjectName,
       RTRIM(cdt.CounterName) AS CounterName,
       MIN(cd.CounterValue) AS "Minimum value",
       MAX(cd.CounterValue) AS "Maximum value",
       AVG(cd.CounterValue) AS "Average value"

  FROM dbo.CounterData cd
    INNER JOIN dbo.DisplayToID did ON cd.GUID = did.GUID
    INNER JOIN dbo.CounterDetails cdt ON cd.CounterID = cdt.CounterID

 WHERE did.DisplayString = 'SQLPerf'

 GROUP BY CONVERT(VARCHAR(10), cd.CounterDateTime, 101),
          RTRIM(cdt.ObjectName), RTRIM(cdt.CounterName)

 ORDER BY RTRIM(cdt.ObjectName), RTRIM(cdt.CounterName)
```

A sample query to display aggregate data from more than one display string:

Aggregate Data for More Than One Display String

```
SELECT did.DisplayString AS DataCollectorSet,
       RTRIM(cdt.ObjectName) AS ObjectName,
       RTRIM(cdt.CounterName) AS CounterName,
       MIN(cd.CounterValue) AS "Minimum value",
       MAX(cd.CounterValue) AS "Maximum value",
       AVG(cd.CounterValue) AS "Average value"

  FROM dbo.CounterData cd
    INNER JOIN dbo.DisplayToID did ON cd.GUID = did.GUID
    INNER JOIN dbo.CounterDetails cdt ON cd.CounterID = cdt.CounterID
 GROUP BY did.DisplayString, RTRIM(cdt.ObjectName), RTRIM(cdt.CounterName)

 ORDER BY RTRIM(cdt.ObjectName), RTRIM(cdt.CounterName);
```

You can use the preceding query to compare counter values from different data collectors. Query results can be written to tables or flat files for further analysis.

Analyzing data manually is relatively straightforward and gives a lot of flexibility with the type of analysis you can perform.

Lab: Monitoring, Tracing, and Baselining

Scenario

You are investigating why a new SQL Server instance is running slowly. Users are complaining that their workloads run particularly slowly during peak business hours. To troubleshoot these performance issues, and take informed corrective measures, you decide to establish a baseline for SQL Server performance.

In this lab exercise, you will set up data collection for analyzing workloads during peak business hours, and implement a baseline methodology to collect performance data at frequent intervals. This will enable comparisons to be made with the baseline.

Objectives

After completing this lab, you will be able to:

- Collect and analyze performance data by using Extended Events.
- Implement a methodology to establish a baseline.

Estimated Time: 60 minutes

Virtual machine: **20762C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Collecting and Analyzing Data Using Extended Events

Scenario

You have been asked to prepare a reusable Extended Events session to collect and analyze workload.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Set Up an Extended Event Session
3. Execute Workload
4. Analyze Collected Data

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab18\Starter** folder as Administrator.

► Task 2: Set Up an Extended Event Session

1. Create an Extended Events session to capture the **sqlserver.error_reported**, **sqlserver.module_end**, **sqlserver.sp_statement_completed**, and **sqlserver.sql_batch_completed** events with a **ring_buffer** target. In the **D:\Labfiles\Lab18\Starter\20762** folder, the **SetupExtendedEvent.sql** file has a possible solution script.
2. Use Watch Live Data for the Extended Events session.

► **Task 3: Execute Workload**

1. In the **D:\Labfiles\Lab18\Starter** folder, in the **RunWorkload.cmd** file, run the workload multiple times to generate event data for the **Extended Event** session.
2. In the **AnalyzeSQLEE** session live data window, stop the feed data, and then add the **duration**, **query_hash**, and **statement** columns to the view.

► **Task 4: Analyze Collected Data**

1. In the **AnalyzeSQLEE** Extended Events live data window, group the data on **query_hash** data, and then aggregate the data on **average of duration**. Sort the data in descending order of duration so that statements that take the highest average time are at the top.
2. Review the data in one of the query hash rows.

Drop the **AnalyzeSQLEE** Extended Events session.

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

Set up an Extended Events session that collects performance data for a workload.

Analyze the data.

Exercise 2: Implementing Baseline Methodology

Scenario

You have been asked to set up a baseline methodology. You must collect data that can be used for comparison if the instance develops performance issues.

The main tasks for this exercise are as follows:

1. Set Up Data Collection Scripts
2. Execute Workload
3. Analyze Data

► **Task 1: Set Up Data Collection Scripts**

- Create a database named **baseline** by using default settings, and then clear the wait statistics for the database. In the **D:\Labfiles\Lab18\Starter\20762** folder, the **PrepareScript.sql** Transact-SQL file has a sample solution script.

► **Task 2: Execute Workload**

1. Create a job from the **WaitsCollectorJob.sql** Transact-SQL file in the **D:\Labfiles\Lab18\Starter\20762** folder.
2. Run the **waits_collectionsjob** to collect statistics before and after running the **RunWorkload.cmd** file multiple times.

► **Task 3: Analyze Data**

1. Using the collected waits data, write and execute a query to find the waits for the workload. In the **D:\Labfiles\Lab18\Starter\20762** folder, the **WaitBaselineDelta.sql** file has a sample solution script.
2. Using the collected waits data, write and execute a query to find the percentage of waits. In the **D:\Labfiles\Lab18\Starter\20762** folder, the **WaitBaselinePercentage.sql** file has a sample solution script.

3. Using the collected waits data, write and execute a query to find the top 10 waits. In the **D:\Labfiles\Lab18\Starter\20762** folder, the **WaitBaselineTop10.sql** file has a sample solution script.
4. Close SQL Server Management Studio without saving any changes.
5. Close File Explorer.

Results: After completing this exercise, you will have implemented a baseline for a workload.

Question: What advantages do you see in using Extended Events to monitor your SQL Server databases?

Module Review and Takeaways

Frequent monitoring and tracing is the key to identifying performance issues. Benchmarks and baselines enable you to implement a robust performance troubleshooting methodology. SQL Server provides a variety of different tools to help you to regularly monitor system performance.

Review Question(s)

Question: Do you use baselines to help you manage your SQL Server systems? If so, what tools do you use to create baselines? If not, what are your reasons for not creating baselines?

Course Evaluation

Course Evaluation

- Your evaluation of this course will help Microsoft understand the quality of your learning experience.
- Please work with your training provider to access the course evaluation form.
- Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.

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

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

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

Module 2: Designing and Implementing Tables

Lab: Designing and Implementing Tables

Exercise 1: Designing Tables

► Task 1: Prepare the Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab02\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

► Task 2: Review the Design

1. In File Explorer, browse to the **D:\Labfiles\Lab02\Starter** folder, and then double-click **Schema Design for Marketing Development Tables.docx**.
2. Review the proposed structure for the three tables.

► Task 3: Improve the Design

1. In the Schema Design document, add Yes or No to the **Allow Nulls?** column for each table, depending whether you think that column should allow nulls.
2. On the **File** menu, click **Save**.
3. On the **File** menu, click **Close**.
4. On the **File** menu, click **Open**.
5. In the **Open** pane, click **Browse**.
6. In the **Open** dialog box, browse to the **D:\Labfiles\Lab02\Solution** folder, click **Schema Design for Marketing Development Tables.docx**, and then click **Open**.
7. Review the suggested nullability for the columns in the three tables.
8. Close WordPad without saving changes.

Results: After completing this exercise, you will have an improved schema and table design.

Exercise 2: Creating Schemas

► Task 1: Create a Schema

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, in the **Authentication** list, click **Windows Authentication**, and then click **Connect**.
3. On the **File** menu, point to **Open**, and then click **Project/Solution**.
4. In the **Open Project** dialog box, navigate to **D:\Labfiles\Lab02\Starter\Project**, and then double-click **Project.ssmssln**.
5. In Solution Explorer, double-click **Lab Exercise 2.sql**.
6. Select the **USE TSQL** statement, and then click **Execute**.
7. Under the comment that starts **Task 1**, type the following query, and then click **Execute**:

```
CREATE SCHEMA DirectMarketing
AUTHORIZATION dbo;
GO
```

8. Leave SQL Server Management Studio open for the next exercise.

Results: After completing this exercise, you will have a new schema in the database.

Exercise 3: Creating Tables

► Task 1: Create the Competitor Table

1. In Solution Explorer, double-click the query **Lab Exercise 3.sql**.
2. Select the **USE TSQL** statement, and then click **Execute**.
3. Under the comment that starts **Task 1**, type the following query, select the query, and then click **Execute**:

```
CREATE TABLE DirectMarketing.Competitor
(
CompetitorCode nvarchar(6) NOT NULL,
Name varchar(30) NOT NULL,
[Address] varchar(max) NULL,
Date_Entered varchar(10) NULL,
Strength_of_competition varchar(8) NULL,
Comments varchar(max) NULL
);
```

4. In Object Explorer, expand **Databases**, expand **TSQL**, expand **Tables**, and verify that the **DirectMarketing.Competitor** table exists.

► **Task 2: Create the TVAdvertisement Table**

- Under the comment that starts **Task 2**, type the following query, select the query, and then click **Execute**:

```
CREATE TABLE DirectMarketing.TVAdvertisement
(
    TV_Station nvarchar(15) NOT NULL,
    City nvarchar(25) NULL,
    CostPerAdvertisement float NULL,
    TotalCostOfAllAdvertisements float NULL,
    NumberOfAdvertisements varchar(4) NULL,
    Date_Of_Advertisement_1 varchar(12) NULL,
    Time_Of_Advertisement_1 int NULL,
    Date_Of_Advertisement_2 varchar(12) NULL,
    Time_Of_Advertisement_2 int NULL,
    Date_Of_Advertisement_3 varchar(12) NULL,
    Time_Of_Advertisement_3 int NULL,
    Date_Of_Advertisement_4 varchar(12) NULL,
    Time_Of_Advertisement_4 int NULL,
    Date_Of_Advertisement_5 varchar(12) NULL,
    Time_Of_Advertisement_5 int NULL
);
GO
```

- In Object Explorer, under **TSQL**, right-click **Tables**, and then click **Refresh**.
- Verify that the **DirectMarketing.TVAdvertisement** table exists.

► **Task 3: Create the CampaignResponse Table**

- Under the comment that starts **Task 3**, type the following query, select the query, and then click **Execute**:

```
CREATE TABLE DirectMarketing.CampaignResponse
(
    ResponseOccurredWhen datetime,
    RelevantProspect int,
    RespondedHow varchar(8),
    ChargeFromReferrer decimal(8,2),
    RevenueFromResponse decimal(8,2),
    ResponseProfit AS (RevenueFromResponse - ChargeFromReferrer) PERSISTED
);
GO
```

- In Object Explorer, under **TSQL**, right-click **Tables**, and then click **Refresh**.
- Verify that the **DirectMarketing.CampaignResponse** table exists.
- Expand **DirectMarketing.CampaignResponse**, and then expand **Columns**. Note that the **ResponseProfit** column is defined as a **Computed** column.
- Double-click **ResponseProfit** and in the **Column Properties - ResponseProfit** dialog box, review the **Computed text** property, and then click **Cancel**.
- Close SQL Server Management Studio without saving changes.

Results: After completing this exercise you will have created the Competitor, TVAdvertisement, and the CampaignResponse tables. You will have created table columns with the appropriate NULL or NOT NULL settings, and primary keys.

Module 3: Advanced Table Designs

Lab: Using Advanced Table Designs

Exercise 1: Partitioning Data

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab03\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then if prompted with the question **Do you want to continue with this operation?**, type **Y**, then press Enter.

► Task 2: Create the HumanResources Database

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** text box, type **MIA-SQL**, and then click **Options**.
3. In the **Connect to database** list, click **<Browse server...>**.
4. In the **Browse for Databases** dialog box, click **Yes**.
5. In the **Browse Server for Database** dialog box, under **User Databases**, click the **TSQL** database, and then click **OK**.
6. In the **Connect to Server** dialog box, on the **Login** tab, in the **Authentication** list, click **Windows Authentication**, and click **Connect**.
7. On the **File** menu, point to **Open**, and click **Project/Solution**.
8. In the **Open Project** dialog box, navigate to **D:\Labfiles\Lab03\Starter\Project\Project**, and then double-click **Project.ssmssln**.
9. In Solution Explorer, double-click the **Lab Exercise 1.sql** query.
10. When the query window opens, review the code, and click **Execute**.

► Task 3: Implement a Partitioning Strategy

1. In Solution Explorer, double-click the query **Lab Exercise 2.sql**.
2. In the query window, under the comment **Create filegroups**, review the Transact-SQL statements, highlight the statements through to the very last line, and then click **Execute**.
3. In the query window, under the comment **Create partition function**, review the Transact-SQL statements, highlight the statements, and then click **Execute**.
4. In the query window, under the comment **Create partition scheme**, review the Transact-SQL statements, highlight the statements, and then click **Execute**.
5. In the query window, under the comment **Create the Timesheet table**, review the Transact-SQL statements, highlight the statements, and then click **Execute**.
6. In the query window, under the comment **Insert data into Timesheet table**, review the Transact-SQL statements, highlight the statements, and then click **Execute**.

► **Task 4: Test the Partitioning Strategy**

1. In Solution Explorer, open the query **Lab Exercise 3.sql**.
2. In the query window, under the comment **Query the Timesheet table**, review the T-SQL statements, highlight the statements, and then click **Execute**.
3. In the query window, under the comment **View partition metadata**, review the T-SQL statements, highlight the statements, and then click **Execute**.
4. In the query window, under the comment **Create the staging table**, review the T-SQL statements, highlight the statements, and then click **Execute**.
5. In the query window, under the comment **Add check constraint**, review the T-SQL statements, highlight the statements, and then click **Execute**.
6. In the query window, under the comment **Switch out the old data**, review the T-SQL statements, highlight the statements, and then click **Execute**.
7. In the query window, under the comment **View archive data in staging table**, review the T-SQL statements, highlight the statements, and then click **Execute**.
8. In the query window, under the comment **View partition metadata**, review the T-SQL statements, highlight the statements, and then click **Execute**.
9. In the query window, under the comment **Merge the first two partitions**, review the T-SQL statements, highlight the statements, and then click **Execute**.
10. In the query window, under the comment **View partition metadata**, review the T-SQL statements, highlight the statements, and then click **Execute**.
11. In the query window, under the comment **Make FG1 the next used filegroup**, review the T-SQL statements, highlight the statements, and then click **Execute**.
12. In the query window, under the comment **Split the empty partition at the end**, review the T-SQL statements, highlight the statements, and then click **Execute**.
13. In the query window, under the comment **Insert data for the new period**, review the T-SQL statements, highlight the statements, and then click **Execute**.
14. In the query window, under the comment **View partition metadata**, review the T-SQL statements, highlight the statements, and then click **Execute**.
15. Leave SQL Server Management Studio open for the next exercise.

Results: At the end of this lab, the timesheet data will be partitioned to archive old data.

Exercise 2: Compressing Data

► Task 1: Create Timesheet Table for Compression

1. In Solution Explorer, double-click the query **Lab Exercise 4.sql**.
2. In the query window, under the comment **Drop the Payment.Timesheet table**, review the Transact-SQL statements, highlight the statement, and then click **Execute**.
3. In the query window, under the comment **Drop the current partition scheme**, review the Transact-SQL statements, highlight the statement, and then click **Execute**.
4. In the query window, under the comment **Drop the current partition function**, review the Transact-SQL statements, highlight the statement, and then click **Execute**.
5. In the query window, under the comment **Create the new partition function with wider date ranges**, review the Transact-SQL statements, highlight the statement, and then click **Execute**.
6. In the query window, under the comment **Re-create the partition scheme**, review the Transact-SQL statements, highlight the statement, and then click **Execute**.
7. In the query window, under the comment **Create the Timesheet table**, review the Transact-SQL statements, highlight the statement, and then click **Execute**.
8. In the query window, under the comment **Insert data into the Payment.Timesheet table**, review the Transact-SQL statements, highlight the statements through to the final line, and then click **Execute**.

► Task 2: Analyze Storage Savings with Compression

1. In Solution Explorer, open the query **Lab Exercise 5.sql**.
2. In the query window, under the comment **View partition metadata**, review the Transact-SQL statements, highlight the statement, and then click **Execute**.
3. In the query window, under the comment **View compression estimated savings**, review the Transact-SQL statements, highlight the statement, and then click **Execute**.

► Task 3: Compress Partitions

1. In Solution Explorer, open the query **Lab Exercise 6.sql**.
2. In the query window, review the Transact-SQL statements, and then click **Execute**.
3. Close SSMS without saving anything.

Results: At the end of this lab, the Timesheet table will be populated with six years of data, and will be partitioned and compressed.

Module 4: Ensuring Data Integrity Through Constraints

Lab: Ensuring Data Integrity Through Constraints

Exercise 1: Add Constraints

► **Task 1: Prepare the Lab Environment**

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. On the taskbar, click **File Explorer**.
3. In File Explorer, navigate to the **D:\Labfiles\Lab04\Starter** folder, right-click the **Setup.cmd** file, and then click **Run as administrator**.
4. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

► **Task 2: Review Supporting Documentation**

- Review the table design requirements that were supplied in the scenario

► **Task 3: Alter the Direct Marketing Table**

1. In File Explorer, navigate to the directory **D:\Labfiles\Lab04\Starter**, double-click on the starter project, which is named **Starter.ssmssqlproj**.
2. In SQL Server Management Studio, in Solution Explorer, in the **Queries** folder, double-click **AlterTable.sql**.
3. Highlight the code below **Step 1: Make sure your database scope is set to Adventureworks**, and click **Execute**.
4. Highlight the code below **Step 2: Explore the metadata for DirectMarketing.Opportunity**, and click **Execute**.
5. Highlight the code below **Step 3: Alter the table to meet the requirements**, and click **Execute**.
6. Under the **Step 4: Add a composite primary key to the table**, type the following code:

```
ALTER TABLE DirectMarketing.Opportunity
ADD CONSTRAINT PK_Opportunity PRIMARY KEY CLUSTERED (OpportunityID, ProspectID);
GO
```

7. Highlight the code and click **Execute**.
8. Under the **Step 5: Add a foreign key to the table, linking it to the Prospect table primary key**, type the following code:

```
ALTER TABLE DirectMarketing.Opportunity
ADD CONSTRAINT FK_OpportunityProspect
FOREIGN KEY (ProspectID) REFERENCES DirectMarketing.Prospect(ProspectID);
GO
```

9. Highlight the code and click **Execute**.

10. Under the **Step 6: Add a default constraint that will set the DateRaised column to the current system data and time**, type the following code:

```
ALTER TABLE DirectMarketing.Opportunity  
ADD CONSTRAINT dfDateRaised  
DEFAULT (SYSDATETIME()) FOR DateRaised;  
GO
```

11. Highlight the code and click **Execute**.
12. Under the **Step 7: Explore the metadata for DirectMarketing.Opportunity**, investigate the changes to the metadata by typing the following code:

```
sp_help 'DirectMarketing.Opportunity'
```

13. Highlight the code and click **Execute**.

Exercise 2: Test the Constraints

► Task 1: Test the Data Types and Default Constraints

1. In Solution Explorer, right-click the **Queries** folder, and then click **New Query**.
2. Right-click the new file, click **Rename**, type **ConstraintTesting.sql**, and then press Enter.
3. In the query pane, type the following query:

```
-- Step 1: Insert a new row to test the data types and default behavior within the  
DirectMarketing.Opportunity table  
INSERT INTO DirectMarketing.Opportunity (OpportunityID, ProspectID,  
Likelihood, Rating, EstimatedClosingDate, EstimatedRevenue)  
VALUES (1,1,8,'A','12/12/2013',123000.00);  
SELECT * FROM DirectMarketing.Opportunity;  
GO
```

4. Highlight the code and click **Execute**.

 **Note:** This query should execute without errors.

► Task 2: Test the Primary Key

1. In the query pane, under the existing code, type the following query:

```
-- Step 2: Try to add the same data again to test that the primary key prevents this  
action  
INSERT INTO DirectMarketing.Opportunity (OpportunityID, ProspectID,  
Likelihood, Rating, EstimatedClosingDate, EstimatedRevenue)  
VALUES (1,1,8,'A','12/12/2013',123000.00);  
GO
```

2. Highlight the code and click **Execute**.

 **Note:** This query should fail due to the PRIMARY KEY constraint.

► **Task 3: Test to Ensure the Foreign Key is Working as Expected**

1. In the query pane, under the existing code, type the following query:

```
-- Step 3: Try to add some data for a prospect that does not exist
INSERT INTO DirectMarketing.Opportunity (OpportunityID,ProspectID,
Likelihood,Rating,EstimatedClosingDate, EstimatedRevenue)
VALUES (2,10,8,'A','12/12/2013',123000.00);
GO
```

2. Highlight the code and click **Execute**.

 **Note:** This query should fail due to the FOREIGN KEY constraint.

3. Close SQL Server Management Studio without saving any changes.

Results: After completing this exercise, you should have successfully tested your constraints.

Module 5: Introduction to Indexes

Lab: Implementing Indexes

Exercise 1: Creating a Heap

► **Task 1: Prepare the Lab Environment**

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab05\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

► **Task 2: Review the Documentation**

1. In File Explorer, navigate to the **D:\Labfiles\Lab05\Starter** folder and then double-click **Supporting Documentation.docx**.
2. Review the requirements in the supporting documentation for the tables, and decide how you are going to meet them.

► **Task 3: Create the Tables**

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, in the **Authentication** list, click **Windows Authentication**, and then click **Connect**.
3. In Object Explorer, expand **Databases**, right-click **AdventureWorks**, and then click **New Query**.
4. Type the following query in the query pane:

```
CREATE TABLE Sales.MediaOutlet (
    MediaOutletID INT NOT NULL,
    MediaOutletName NVARCHAR(40),
    PrimaryContact NVARCHAR (50),
    City NVARCHAR (50)
);
```

5. On the toolbar, click **Execute**.
6. In Object Explorer, right-click **AdventureWorks**, and in the context menu click **New Query**.
7. Type the following query in the query pane:

```
CREATE TABLE Sales.PrintMediaPlacement ( PrintMediaPlacementID INT NOT NULL,
    MediaOutletID INT,
    PlacementDate DATETIME,
    PublicationDate DATETIME,
    RelatedProductID INT,
    PlacementCost DECIMAL(18,2)
);
```

8. On the toolbar, click **Execute**.
9. In Object Explorer, expand **AdventureWorks**, and then expand **Tables**. Note that your two new tables are included in the list. If no new tables appear, in Object Explorer, click **Refresh**.
10. Expand **Sales.MediaOutlet**, expand **Indexes**, and note that there are no indexes on the table.

11. Repeat Step 10 for the **Sales.PrintMediaPlacement** table.
12. Leave SQL Server Management Studio open for the next exercise.

Results: After completing this exercise, you will have created two new tables in the AdventureWorks database.

Exercise 2: Creating a Clustered Index

► Task 1: Add a Clustered Index to Sales.MediaOutlet

1. In Object Explorer, right-click **AdventureWorks**, and then click **New Query**.
2. Type the following query in the query pane:

```
ALTER TABLE Sales.MediaOutlet ADD CONSTRAINT IX_MediaOutlet UNIQUE CLUSTERED (
    MediaOutletID
);
```
3. On the toolbar, click **Execute**.
4. In Object Explorer, click **Sales.MediaOutlet**, and then on the toolbar, click **Refresh**.
5. Expand **Indexes** and note that the table has a clustered index named **IX_MediaOutlet (Clustered)**.
6. Expand **Keys** and note that the table has a key named **IX_MediaOutlet**.

► Task 2: Add a Clustered Index to Sales.PrintMediaPlacement

1. In Object Explorer, right-click **AdventureWorks**, and then click **New Query**.
2. Type the following query in the query pane:

```
CREATE UNIQUE CLUSTERED INDEX CIX_PrintMediaPlacement ON Sales.PrintMediaPlacement (
    PrintMediaPlacementID ASC
);
```
3. On the toolbar, click **Execute**.
4. In Object Explorer, click **Sales.PrintMediaPlacement**, and then on the toolbar, click **Refresh**.
5. Expand **Indexes** and note that the table has a clustered index named **CIX_PrintMediaPlacement (Clustered)**.
6. Leave SQL Server Management Studio open for the next exercise.

Results: After completing this exercise, you will have created clustered indexes on the new tables.

Exercise 3: Creating a Covering Index

► Task 1: Add Some Test Data

1. On the **File** menu, point to **Open**, and then click **File**.
2. In the **Open File** dialog box, navigate to the **D:\Labfiles\Lab05\Starter** folder, click **InsertDummyData.sql**, and then click **Open**.
3. On the toolbar, click **Execute**.

► Task 2: Run the Poor Performing Query

1. On the **File** menu, point to **Open**, and then click **File**.
2. In the **Open File** dialog box, navigate to the **D:\Labfiles\Lab05\Starter** folder, click **SalesQuery.sql**, and then click **Open**.
3. On the **Query** menu, click **Include Actual Execution Plan**.
4. On the toolbar, click **Execute**.
5. On the **Execution Plan** tab, note the missing index warning.

► Task 3: Create a Covering Index

1. On the **Execution Plan** tab, move the mouse over the green **Missing Index** text.
2. Right-click the green **Missing Index** text, and then click **Missing Index Details**.
3. In the query pane, delete the `/*` on line 6 and then delete the `*/` on line 13.
4. Modify the query so that it reads as follows:

```
USE [AdventureWorks]
GO
CREATE NONCLUSTERED INDEX NCI_PrintMediaPlacement
ON [Sales].[PrintMediaPlacement] ([PublicationDate],[PlacementCost])
INCLUDE ([PrintMediaPlacementID],[MediaOutletID],[PlacementDate],[RelatedProductID])
GO
```

5. On the toolbar, click **Execute**.
6. In Object Explorer, click **Sales.PrintMediaPlacement**, and then on the toolbar, click **Refresh**.
7. Expand **Indexes** and note that there is a nonclustered index named **NCI_PrintMediaPlacement (Non-Unique, Non-Clustered)**.

► Task 4: Check the Performance of the Sales Query

1. Switch to the **SalesQuery.sql** query file, and on the toolbar, click **Execute**.
2. On the **Execution Plan** tab, note that SQL Server Management Studio no longer warns of a missing index.
3. Note that the new **NCI_PrintMediaPlacement** index is being used.
4. Close SQL Server Management Studio without saving any changes.

Results: After completing this exercise, you will have created a covering index suggested by SQL Server Management Studio.

Module 6: Designing Optimized Index Strategies

Lab: Optimizing Indexes

Exercise 1: Using Query Store

► Task 1: Use Query Store to Monitor Query Performance

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are running.
2. Log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
3. In the **D:\Labfiles\Lab06\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
4. In the **User Account Control** dialog box, click **Yes**, and wait for the script to finish.
5. Start SSMS and connect to the **MIA-SQL** database engine instance using Windows authentication.
6. On the **File** menu, point to **Open**, and then click **File**.
7. In the **Open File** dialog box, navigate to **D:\Labfiles\Lab06\Starter**, click **QueryStore_Lab1.sql**, and then click **Open**.
8. Select the code under the comment **Use the AdventureWorks2016 database**, and then click **Execute**.
9. Select the code under the comment **Create indexed view**, and then click **Execute**.
10. Select the code under the comment **Clear the Query Store**, and then click **Execute**.
11. Select the code under the comment **Run a select query six times**, and then click **Execute**.
12. Repeat the last step another five times, waiting a few seconds between each execution.
13. Select the code under the comment **Update the statistics with fake figures**, and then click **Execute**.
14. Repeat Step 11 twice, waiting a few seconds between each execution.
15. In Object Explorer, expand the **Databases** node, expand **AdventureWorks2016**, expand **Query Store**, and double-click **Top Resource Consuming Queries**.
16. Examine the different views of the Top 25 Resources report.
17. Locate the original query plan, and click **Force Plan**.
18. In the **Confirmation** dialog box, click **Yes**.
19. Switch to the **QueryStore_Lab1.sql** tab.
20. Repeat Step 11 three times, waiting a few seconds between each execution.
21. Switch to the **Top Resource Consuming Queries** tab.
22. Using the different views of the report, identify which query plans used a clustered index seek and which ones used a clustered index scan.
23. Keep SSMS open for the next lab exercise.

Results: After completing this lab exercise you will have used Query Store to monitor query performance, and used it to force a particular execution plan to be used.

Exercise 2: Heaps and Clustered Indexes

► Task 1: Compare a Heap with a Clustered Index

1. On the **File** menu, point to **Open**, and then click **File**.
2. In the **Open File** dialog box, navigate to **D:\Labfiles\Lab06\Starter**, click **ClusterVsHeap_lab.sql**, and then click **Open**.
3. Select the code under the comment **ClusterVsHeap_lab**, and then click **Execute** to make **AdventureWorks2016** the current database.
4. Select the code under the comment **Create a heap and a clustered index**, and then click **Execute**.
5. Select the code under the comment **SET STATISTICS ON**, and then click **Execute**. The script includes a number of different select statements that will be run on both the heap, and the clustered index.
6. Using file explorer, navigate to **D:\Labfiles\Lab06\Starter** and open **HeapVsClustered_Timings.docx**. Use the document to note the CPU time whilst running the code in the following steps.
7. In SSMS, select the code under the comment **SELECT**, and then click **Execute**.
8. On the **Messages** tab, note the CPU time.
9. Select the code under the comment **SELECT ORDER BY**, and then click **Execute**.
10. On the **Messages** tab, note the CPU time.
11. Select the code under the comment **SELECT WHERE**, and then click **Execute**.
12. On the **Messages** tab, note the CPU time.
13. Select the code under the comment **SELECT WHERE ORDER BY**, and then click **Execute**.
14. On the **Messages** tab, note the CPU time.
15. Select the code under the comment **INSERT**, and then click **Execute**.
16. On the **Messages** tab, note the CPU time.
17. Select the code under the comment **SELECT ORDER BY**, and then click **Execute**.
18. On the **Messages** tab, note the CPU time.
19. Compare your results with the timings in the **D:\Labfiles\Lab06\Solution\HeapVsClustered_Solution.docx** file.
20. If you have time, run the select statements again with Live Query Statistics. Click the **Include Live Query Statistics** button on the toolbar, and run each query again.
21. Close SQL Server Management Studio, without saving any changes.
22. Close Wordpad.

Results: After completing this lab exercise, you will:

Understand the effect of adding a clustered index to a table.

Understand the performance difference between a clustered index and a heap.

Module 7: Columnstore Indexes

Lab: Using Columnstore Indexes

Exercise 1: Create a Columnstore Index on the FactProductInventory Table

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Using File Explorer, navigate to the **D:\Labfiles\Lab07\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. When you are prompted, click **Yes** to confirm that you want to run the command file, and then wait for the script to finish.
4. Press any key to continue.

► Task 2: Examine the Existing Size of the FactProductInventory Table and Query Performance

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** window, ensure the **Server name** box contains **MIA-SQL** and the **Authentication** box contains **Windows Authentication**, and then click **Connect**.
3. On the **File** menu, point to **Open**, then click **File**.
4. In the **Open File** dialog box, navigate to the **D:\Labfiles\Lab07\Starter** folder, click **Query FactProductInventory.sql**, and then click **Open**.
5. Highlight the Transact-SQL under the Step 1 comment, and then click **Execute**.
6. Make a note of the disk space used.
7. On the **Query** menu, click **Include Actual Execution Plan**.
8. Highlight the code after the Step 2 comment, click **Execute**, and then view the query results.
9. On the **Messages** tab, note the CPU and elapsed times.
10. On the **Execution plan** tab, view the execution plan for the query. Position the cursor on the icons from right to left to examine their details, and then note the indexes used. Also, note that the query processor has identified that adding a missing index could improve performance.

► Task 3: Create a Columnstore Index on the FactProductInventory Table

1. On the toolbar, click **New Query**, and then in the SQLQuery1.sql pane, type the following Transact-SQL code to create a columnstore index on the **FactProductInventory** table:

```
CREATE NONCLUSTERED COLUMNSTORE INDEX NCI_FactProductInventory_UnitCost_UnitsOut ON
FactProductInventory
(
    ProductKey,
    DateKey,
    UnitCost,
    UnitsOut
)
GO
```

2. On the toolbar, click **Execute** to create the index.

3. Switch back to the **Query FactProductInventory.sql** tab, and then click **Execute** to rerun the query.
4. On the **Execution plan** tab, view the execution plan that is used for the query. Examine the icons from right to left, noting the indexes that were used. Note that the new columnstore index is used.
5. Note the **Actual Execution Mode** is **Batch** instead of **Row** and that this mode is on most of the steps. This is the main reason for the query performance improvement.
6. What, if any, are the disk space and query performance improvements?
7. Close both query windows without saving any changes.

Results: After completing this exercise, you will have created a columnstore index and improved the performance of an analytical query. This will have been done in real time without impacting transactional processing.

Exercise 2: Create a Columnstore Index on the FactInternetSales Table

► Task 1: Examine the Existing Size of the FactInternetSales Table and Query Performance

1. On the **File** menu, point to **Open**, then click **File**.
2. In the **Open File** dialog box, navigate to the **D:\Labfiles\Lab07\Starter** folder, click **Query FactInternetSales.sql**, and then click **Open**.
3. Highlight the Transact-SQL under the Step 1 comment, and then click **Execute**.
4. Make a note of the disk space used.
5. On the **Query** menu, click **Include Actual Execution Plan**.
6. Highlight the code after the Step 2 comment.
7. On the toolbar, click **Execute**, and then view the query results.
8. On the **Messages** tab, note the CPU and elapsed times.
9. On the **Execution** plan tab, view the execution plan for the query. Position the cursor on the icons from right to left to examine their details and note the indexes that were used. Also note that the query processor has identified that adding a missing index could improve performance.

► Task 2: Create a Columnstore Index on the FactInternetSales Table

1. On the **File** menu, point to **Open**, then click **File**.
2. In the **Open File** dialog box, navigate to the **D:\Labfiles\Lab07\Starter** folder, click **Drop Indexes on FactInternetSales.sql**, and then click **Open**.
3. Click **Execute** to drop the existing indexes.
4. On the **File** menu, point to **Open**, then click **File**.
5. Click **Create Columnstore Index on FactInternetSales.sql**, and then click **Open**.
6. Click **Execute** to drop the existing indexes.
7. Switch back to the **Query FactInternetSales.sql** tab, and then click **Execute** to rerun the query.

8. On the **Execution plan** tab, scroll to the end to view the execution plan that is used for the query. Examine the icons from right to left, noting the indexes that were used. Note that the new columnstore index is used.
9. What, if any, are the disk space and query performance improvements?
10. Close all the query windows without saving changes.

Results: After completing this exercise, you will have greatly reduced the disk space taken up by the FactInternetSales table, and improved the performance of analytical queries against the table.

Exercise 3: Create a Memory Optimized Columnstore Table

► Task 1: Use the Memory Optimization Advisor

1. In SQL Server Management Studio, in **Object Explorer**, expand **Databases**, expand **AdventureWorksDW**, and then expand **Tables**.
2. Right-click **dbo.FactInternetSales**, and then click **Memory Optimization Advisor**.
3. In the **Table Memory Optimization Advisor** window, on the **Introduction** page, click **Next**.
4. On the **Migration validation** page, scroll down, looking at the descriptions against any row in the table without a green tick.
5. Note that the advisor requires that the foreign key relationships are removed before it can continue.
6. Note that the advisor also requires the removal of the clustered columnstore index.
7. Click **Cancel**.

► Task 2: Enable the Memory Optimization Advisor to Create a Memory Optimized FactInternetSales Table

1. On the **File** menu, point to **Open**, then click **File**.
2. In the **Open File** dialog box, navigate to the **D:\Labfiles\Lab07\Starter** folder, click **Drop Columnstore Indexes on FactInternetSales.sql**, and then click **Open**.
3. Click **Execute** to drop the existing columnstore index and foreign keys.
4. In Object Explorer, right-click **dbo.FactInternetSales**, and then click **Memory Optimization Advisor**.
5. In the **Table Memory Optimization Advisor** window, on the **Introduction** page, click **Next**.
6. On the **Migration validation** page, click **Next**.
7. On the **Memory Optimization Warnings** page, note the warnings and click **Next**.
8. On the **Migration options** page, select the **Also copy table data to the new memory optimized table** check box, and then click **Next**.
9. On the **Index creation** page, in the column list, select the **SalesOrderNumber** and **SalesOrderLineNumber** check boxes, and then click **Next**.
10. On the **Summary** page, click **Script**.
11. Inspect the generated Transact-SQL. Note the code to create the memory optimized filegroup and the renaming of the existing table.



Note: The Memory Optimization Advisor won't suggest columnstore indexes as they are not applicable in all situations. Therefore, you have to add these manually.

12. Locate the following lines of Transact-SQL:

```
)WITH ( BUCKET_COUNT = 2097152)
)WITH ( MEMORY_OPTIMIZED = ON , DURABILITY = SCHEMA_AND_DATA )
```

13. Add the following Transact-SQL between the two WITH statements:

```
, INDEX CCI_OnlineFactInternetSales CLUSTERED COLUMNSTORE
```

14. On the toolbar, click **Execute**.

► **Task 3: Examine the Performance of the Memory Optimized Table**

1. On the **File** menu, point to **Open**, and then click **File**.
2. In the **Open File** dialog box, navigate to the **D:\Labfiles\Lab07\Starter** folder, click **Query FactInternetSales.sql**, and then click **Open**.
3. Highlight the Transact-SQL under the Step 1 comment, and then on the toolbar, click **Execute**.
4. Note that no disk space is used, as this table is now memory optimized.
5. On the **Query** menu, click **Include Actual Execution Plan**.
6. Highlight the code after the Step 2 comment, click **Execute**, and then view the query results.
7. On the **Execution plan** tab, view the execution plan for the query, and note the indexes that were used. Position the cursor on the icons to examine their details.
8. Close SQL Server Management Studio without saving changes.

Results: After completing this exercise, you will have created a memory optimized version of the **FactInternetSales** disk based table, using the Memory Optimization Advisor.

Module 8: Designing and Implementing Views

Lab: Designing and Implementing Views

Exercise 1: Creating Standard Views

► **Task 1: Prepare the Environment**

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab08\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

► **Task 2: Design and Implement the Views**

1. Review the documentation for the views.
2. On the taskbar, click **Microsoft SQL Server Management Studio**.
3. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, in the **Authentication** list, click **Windows Authentication**, and then click **Connect**.
4. On the toolbar, click **New Query**.
5. In the new query pane, type the following code to create the **Production.OnlineProducts** view:

```
USE AdventureWorks2016;
GO
CREATE VIEW
Production.OnlineProducts
AS
SELECT p.ProductID, p.Name, p.ProductNumber AS [Product Number], COALESCE(p.Color,
'N/A') AS Color,
CASE p.DaysToManufacture
WHEN 0 THEN 'In stock'
WHEN 1 THEN 'Overnight'
WHEN 2 THEN '2 to 3 days delivery'
ELSE 'Call us for a quote'
END AS Availability,
p.Size, p.SizeUnitMeasureCode AS [Unit of Measure], p.ListPrice AS Price, p.Weight
FROM Production.Product AS p
WHERE p.SellEndDate IS NULL AND p.SellStartDate IS NOT NULL;
GO
```

6. Below the query that you have just typed, type the following code to create the **Production.AvailableModels** view:

```
USE AdventureWorks2016;
GO
CREATE VIEW
Production.AvailableModels
AS
SELECT p.ProductID AS [Product ID], p.Name, pm.ProductModelID AS [Product Model ID],
pm.Name as [Product Model]
FROM Production.Product AS p
INNER JOIN Production.ProductModel AS pm
ON p.ProductModelID = pm.ProductModelID
WHERE p.SellEndDate IS NULL
AND p.SellStartDate IS NOT NULL;
```

```
GO
```

7. On the toolbar, click **Execute** to create the views.

► Task 3: Test the Views

1. On the **File** menu, point to **New**, and then click **Query with Current Connection**.
2. In the new query pane, type the following code to test the new views:

```
USE AdventureWorks2016;
GO
SELECT * FROM Production.OnlineProducts;
GO
SELECT * FROM Production.AvailableModels;
GO
```

3. On the toolbar, click **Execute**.
4. Check that each view is returning the correct columns, check that the column headings are correct, and check that only products that are available to be sold are listed.
5. Leave SSMS open for use in the next exercise.

Results: After completing this exercise, you will have two new views in the AdventureWorks database.

Exercise 2: Creating an Updateable View

► Task 1: Design and Implement the Updateable View

1. Review the requirements for the updateable view.
2. In SSMS, on the **File** menu, point to **New**, and then click **Query with Current Connection**.
3. In the new query pane, type the following code to create the updateable **Sales.NewCustomer** view:

```
USE AdventureWorks2016;
GO
CREATE VIEW Sales.NewCustomer
AS
SELECT CustomerID, FirstName, LastName
FROM Sales.CustomerPII;
GO
```

4. On the toolbar, click **Execute** to create the view.

► Task 2: Test the Updateable View

1. On the **File** menu, point to **New**, and then click **Query with Current Connection**.
2. In the new query pane, type the following code to test that the new view returns the correct data:

```
USE AdventureWorks2016;
GO
SELECT * FROM Sales.NewCustomer
ORDER BY CustomerID
```

3. On the toolbar, click **Execute**.

4. Check that the view is returning the correct columns.
5. Above the query that you have just typed, type the following code to test that the new view is updateable:

```
USE AdventureWorks2016;
GO
INSERT INTO Sales.NewCustomer
VALUES
(10001, 'Ed', 'Kish'),
(10002, 'Kermit', 'Albritton');
GO
```

6. On the toolbar, click **Execute**.
7. Check that the new rows now appear in the results set from the view.
8. Close SSMS without saving any changes.

Results: After completing this exercise, you will have a new updateable view in the database.

Module 9: Designing and Implementing Stored Procedures

Lab: Designing and Implementing Stored Procedures

Exercise 1: Create Stored Procedures

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
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 wait for the script to finish.

► Task 2: Review the Reports.GetProductColors Stored Procedure Specification

- Review the design requirements in the Exercise Scenario for Marketing.GetProductColors.

► Task 3: Design, Create and Test the Reports.GetProductColors Stored Procedure

1. Start SQL Server Management Studio, and connect to the **MIA-SQL** database instance using Windows authentication.
2. In Object Explorer, expand **MIA-SQL**, and then expand **Databases**.
3. Right-click the **MarketDev** database, and then click **New Query**.
4. In the query pane, type the query below.

```
CREATE PROCEDURE Reports.GetProductColors
AS
SET NOCOUNT ON;
BEGIN
SELECT DISTINCT p.Color
FROM Marketing.Product AS p
WHERE p.Color IS NOT NULL
ORDER BY p.Color;
END
GO
EXEC Reports.GetProductColors;
GO
```

5. On the toolbar, click **Execute**.
6. **Note:** Ensure that approximately nine colors are returned and that no NULL row is returned.

► Task 4: Review the Reports.GetProductsAndModels Stored Procedure Specification

7. Review the supplied design requirements in the supporting documentation in the Exercise Scenario for Reports.GetProductsAndModels.

► **Task 5: Design, Create and Test the Reports.GetProductsAndModels Stored Procedure**

1. In Object Explorer, right-click the **MarketDev** database and then click **New Query**.
2. In the query pane, type the following query:

```
CREATE PROCEDURE Reports.GetProductsAndModels
AS
SET NOCOUNT ON;
BEGIN
SELECT p.ProductID,
       p.ProductName,
       p.ProductNumber,
       p.SellStartDate,
       p.SellEndDate,
       p.Color,
       pm.ProductModelID,
       COALESCE(ed.Description,id.Description,p.ProductName) AS EnglishDescription,
       COALESCE(fd.Description,id.Description,p.ProductName) AS FrenchDescription,
       COALESCE(cd.Description,id.Description,p.ProductName) AS ChineseDescription
FROM Marketing.Product AS p
LEFT OUTER JOIN Marketing.ProductModel AS pm
ON p.ProductModelID = pm.ProductModelID
LEFT OUTER JOIN Marketing.ProductDescription AS ed
ON pm.ProductModelID = ed.ProductModelID
AND ed.LanguageID = 'en'
LEFT OUTER JOIN Marketing.ProductDescription AS fd
ON pm.ProductModelID = fd.ProductModelID
AND fd.LanguageID = 'fr'
LEFT OUTER JOIN Marketing.ProductDescription AS cd
ON pm.ProductModelID = cd.ProductModelID
AND cd.LanguageID = 'zh-cht'
LEFT OUTER JOIN Marketing.ProductDescription AS id
ON pm.ProductModelID = id.ProductModelID
AND id.LanguageID = ''
ORDER BY p.ProductID,pm.ProductModelID;
END
GO
EXEC Reports.GetProductsAndModels;
GO
```

3. On the toolbar, click **Execute**.

Results: After completing this lab, you will have created and tested two stored procedures, Reports.GetProductColors and Reports.GetProductsAndModels.

Exercise 2: Create Parameterized Stored Procedures

- ▶ **Task 1: Review the Reports.GetProductsByColor Stored Procedure Specification**
 - Review the design requirements in the Exercise Scenario for Reports.GetProductsByColor.
- ▶ **Task 2: Design, Create and Test the Reports.GetProductsByColor Stored Procedure**
 1. In Object Explorer, right-click the **MarketDev** database and then click **New Query**.
 2. In the query pane, type the following query:

```
CREATE PROCEDURE Marketing.GetProductsByColor
@Color nvarchar(16)
AS
SET NOCOUNT ON;
BEGIN
SELECT p.ProductID,
p.ProductName,
p.ListPrice AS Price,
p.Color,
p.Size,
p.SizeUnitMeasureCode AS UnitOfMeasure
FROM Marketing.Product AS p
WHERE (p.Color = @Color) OR (p.Color IS NULL AND @Color IS NULL)
ORDER BY ProductName;
END
GO
```

3. On the toolbar, click **Execute**.
4. In the query pane, under the existing code, type the following Transact-SQL, highlight the query, and then click **Execute**:

```
EXEC Marketing.GetProductsByColor 'Blue';
GO
EXEC Marketing.GetProductsByColor NULL;
GO
```

 **Note:** Ensure that approximately 26 rows are returned for blue products. Ensure that approximately 248 rows are returned for products that have no color.

Results: After completing this exercise, you will have:

Created the GetProductsByColor parameterized stored procedure.

Exercise 3: Change Stored Procedure Execution Context

► Task 1: Alter the Reports.GetProductColors Stored Procedure to Execute As OWNER

1. In Object Explorer, right-click the **MarketDev** database and then click **New Query**.
2. In the query pane, type the following query:

```
ALTER PROCEDURE Reports.GetProductColors
WITH EXECUTE AS OWNER
AS
SET NOCOUNT ON;
BEGIN
SELECT DISTINCT p.Color
FROM Marketing.Product AS p
WHERE p.Color IS NOT NULL
ORDER BY p.Color;
END
GO
```

3. On the toolbar, click **Execute**.

► Task 2: Alter the Reports.GetProductsAndModels Stored Procedure to Execute As OWNER.

1. In Object Explorer, right-click the **MarketDev** database and then click **New Query**.
2. In the query pane, type the following query:

```
ALTER PROCEDURE Reports.GetProductsAndModels
WITH EXECUTE AS OWNER
AS
SET NOCOUNT ON;
BEGIN
SELECT p.ProductID,
       p.ProductName,
       p.ProductNumber,
       p.SellStartDate,
       p.SellEndDate,
       p.Color,
       pm.ProductModelID,
       COALESCE(ed.Description,id.Description,p.ProductName) AS EnglishDescription,
       COALESCE(fd.Description,id.Description,p.ProductName) AS FrenchDescription,
       COALESCE(cd.Description,id.Description,p.ProductName) AS ChineseDescription
FROM Marketing.Product AS p
LEFT OUTER JOIN Marketing.ProductModel AS pm
ON p.ProductModelID = pm.ProductModelID
LEFT OUTER JOIN Marketing.ProductDescription AS ed
ON pm.ProductModelID = ed.ProductModelID
AND ed.LanguageID = 'en'
LEFT OUTER JOIN Marketing.ProductDescription AS fd
ON pm.ProductModelID = fd.ProductModelID
AND fd.LanguageID = 'fr'
LEFT OUTER JOIN Marketing.ProductDescription AS cd
ON pm.ProductModelID = cd.ProductModelID
AND cd.LanguageID = 'zh-cht'
LEFT OUTER JOIN Marketing.ProductDescription AS id
ON pm.ProductModelID = id.ProductModelID
AND id.LanguageID = ''
ORDER BY p.ProductID,pm.ProductModelID;
END
GO
```

3. On the toolbar, click **Execute**.

► **Task 3: Alter the Reports.GetProductsByColor Stored Procedure to Execute As OWNER**

1. In Object Explorer, right-click the **MarketDev** database, and then click **New Query**.
2. In the query pane, type the following query:

```
ALTER PROCEDURE Marketing.GetProductsByColor
@Color nvarchar(16)
WITH EXECUTE AS OWNER
AS
SET NOCOUNT ON;
BEGIN
SELECT p.ProductID,
p.ProductName,
p.ListPrice AS Price,
p.Color,
p.Size,
p.SizeUnitMeasureCode AS UnitOfMeasure
FROM Marketing.Product AS p
WHERE (p.Color = @Color) OR (p.Color IS NULL AND @Color IS NULL)
ORDER BY ProductName;
END
GO
```

3. On the toolbar, click **Execute**.

Results: After completing this exercise, you will have altered the three stored procedures created in earlier exercises, so that they run as owner.

Module 10: Designing and Implementing User-Defined Functions

Lab: Designing and Implementing User-Defined Functions

Exercise 1: Format Phone Numbers

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab10\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Access Control** dialog box, click **Yes**.
4. Wait for **setup.cmd** to complete successfully.

► Task 2: Review the Design Requirements

1. In File Explorer, navigate to **D:\Labfiles\Lab10\Starter**, and then double-click **Supporting Documentation.docx**.
2. Review the **Function Specifications: Phone Number** section in the supporting documentation.

► Task 3: Design and Create the Function

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, ensure the server name is **MIA-SQL**, and then click **Connect**.
3. In Object Explorer, expand **MIA-SQL**, expand **Databases**, right-click **AdventureWorks**, and then click **New Query**.
4. Type the following query in the query pane:

```

CREATE FUNCTION dbo.FormatPhoneNumber
( @PhoneNumberToFormat nvarchar(16)
)
RETURNS nvarchar(16)
AS BEGIN
DECLARE @Digits nvarchar(16) = '';
DECLARE @Remaining nvarchar(16) = @PhoneNumberToFormat;
DECLARE @Character nchar(1);
IF LEFT(@Remaining,1) = N'+' RETURN @Remaining;
WHILE (LEN(@Remaining) > 0) BEGIN
SET @Character = LEFT(@Remaining,1);
SET @Remaining = SUBSTRING(@Remaining,2,LEN(@Remaining));
IF (@Character >= N'0') AND (@Character <= N'9')
SET @Digits += @Character;
END;
RETURN CASE LEN(@Digits)
WHEN 10 THEN N'(' + SUBSTRING(@Digits,1,3) + N') '
+ SUBSTRING(@Digits,4,3) + N'-' 
+ SUBSTRING(@Digits,7,4)
WHEN 8 THEN SUBSTRING(@Digits,1,4) + N'-' 
+ SUBSTRING(@Digits,5,4)
WHEN 6 THEN SUBSTRING(@Digits,1,3) + N'-' 
+ SUBSTRING(@Digits,4,3)

```

```
ELSE @Digits
END;
END;
GO
```

5. In the toolbar, click **Execute**.

► Task 4: Test the Function

1. In Object Explorer, right-click **AdventureWorks**, and then click **New Query**.
2. In the query pane, type the following query:

```
SELECT dbo.FormatPhoneNumber('+61 3 9485-2342');
SELECT dbo.FormatPhoneNumber('415 485-2342');
SELECT dbo.FormatPhoneNumber('(41) 5485-2342');
SELECT dbo.FormatPhoneNumber('94852342');
SELECT dbo.FormatPhoneNumber('85-2342');
GO
```

3. In the toolbar, click **Execute**.



Note:

The output should resemble the following:

```
+61 3 9485-2342
(415) 485-2342
(415) 485-2342
9485-2342
```

4. 852-342

Results: After this exercise, you should have created a new **FormatPhoneNumber** function within the **dbo** schema.

Exercise 2: Modify an Existing Function

► Task 1: Review the Requirements

- In WordPad, in the **Supporting Documentation.docx**, review the requirement for the **dbo.IntegerListToTable** function in the supporting documentation.

► Task 2: Design and Create the Function

- In SQL Server Management Studio, in Object Explorer, right-click **AdventureWorks**, and then click **New Query**.
- In the query pane, type the following query:

```
CREATE FUNCTION dbo.IntegerListToTable
( @InputList nvarchar(MAX) ,@Delimiter nchar(1) = N',' )
RETURNS @OutputTable TABLE (PositionInList int IDENTITY(1, 1) NOT NULL, IntegerValue
int)
AS BEGIN
    INSERT INTO @OutputTable (IntegerValue)
    SELECT Value FROM STRING_SPLIT ( @InputList , @Delimiter );
    RETURN;
END;
GO
```

- In the toolbar, click **Execute**.

► Task 3: Test the Function

- In Object Explorer, right-click the **AdventureWorks** database, and then click **New Query**.
- In the query pane, type the following query:

```
SELECT * FROM dbo.IntegerListToTable('234,354253,3242,2', ',' );
GO
```

- In the toolbar, click **Execute**.

Note that the output should resemble the following:

PositionInList	IntegerValue
1	234
2	354253
3	3242
4	2

► Task 4: Test the Function by Using an Alternate Delimiter

- In Object Explorer, right-click the **AdventureWorks** database, and then click **New Query**.
- In the query pane, type the following query:

```
SELECT * FROM dbo.IntegerListToTable('234|354253|3242|2', '|');
GO
```

3. In the toolbar, click **Execute**.

Note that the output should resemble the following:

PositionInList	IntegerValue
1	234
2	354253
3	3242
4	2

4. Close SSMS without saving.

Results: After this exercise, you should have created a new **IntegerListToTable** function within a dbo schema.

Module 11: Responding to Data Manipulation Via Triggers

Lab: Responding to Data Manipulation by Using Triggers

Exercise 1: Create and Test the Audit Trigger

► **Task 1: Prepare the Lab Environment**

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab11\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Access Control** dialog box, click **Yes** and wait for the script to complete.

► **Task 2: Review the Design Requirements**

1. Using File Explorer, browse to **D:\Labfiles\Lab11\Starter**.
2. Double-click **Supporting Documentation.docx**, review the supplied table requirements in the supporting documentation for the **Production.ProductAudit** table.
3. On the taskbar, click **Microsoft SQL Server Management Studio**.
4. In the **Connect to Server** dialog box, ensure that the **Server name** is **MIA-SQL**, and then click **Connect**.
5. In Object Explorer, under **MIA-SQL**, expand **Databases**, expand **AdventureWorks**, expand **Tables**, expand **Production.Product**, and then expand **Columns**.
6. Review the table design.
7. Do not close SQL Server Management Studio.

► **Task 3: Design a Trigger**

1. Click **New Query**.
2. In the query pane, type the following query:

```

USE AdventureWorks
GO
CREATE TABLE Production.ProductAudit(
    ProductID int NOT NULL,
    UpdateTime datetime2(7) NOT NULL,
    ModifyingUser nvarchar(100) NOT NULL,
    OriginalListPrice money NULL,
    NewListPrice money NULL
)
GO
CREATE TRIGGER Production.TR_ProductListPrice_Update
ON Production.Product
AFTER UPDATE
AS BEGIN
    SET NOCOUNT ON;
    INSERT Production.ProductAudit(ProductID, UpdateTime, ModifyingUser,
    OriginalListPrice,NewListPrice)
    SELECT Inserted.ProductID,SYSDATETIME(),ORIGINAL_LOGIN(),deleted.ListPrice,
    inserted.ListPrice
    FROM deleted

```

```
INNER JOIN inserted
    ON deleted.ProductID = inserted.ProductID
    WHERE deleted.ListPrice > 1000 OR inserted.ListPrice > 1000;
END;
GO
```

3. On the toolbar, click **Execute**.
4. Do not close SQL Server Management Studio.

► **Task 4: Test the Behavior of the Trigger**

1. Click **New Query**.
2. In the query pane, type the following query:

```
UPDATE Production.Product
SET ListPrice=3978.00
WHERE ProductID BETWEEN 749 and 753;
GO
SELECT * FROM Production.ProductAudit;
GO
```

3. On the toolbar, click **Execute**.
4. Note: there should be five rows in the **Production.ProductAudit** table.
5. Do not close SQL Server Management Studio.

Results: After this exercise, you should have created a new trigger. Tests should have shown that it is working as expected.

Exercise 2: Improve the Audit Trigger

► Task 1: Modify the Trigger

1. In Object Explorer, under **Databases**, right-click the **MarketDev** database, and then click **Refresh**.
2. Expand the **MarketDev** database, expand **Tables**, expand **Marketing.CampaignBalance**, and then expand **Triggers**.
3. Double-click **TR_CampaignBalance_Update**, and review the trigger design in the query pane.
4. Click **New Query**.
5. In the query pane, type the following query:

```
USE MarketDev
GO
ALTER TRIGGER Marketing.TR_CampaignBalance_Update
ON Marketing.CampaignBalance
AFTER UPDATE
AS BEGIN
SET NOCOUNT ON;
INSERT Marketing.CampaignAudit
(AuditTime, ModifyingUser, RemainingBalance)
SELECT SYSDATETIME(),
ORIGINAL_LOGIN(),
inserted.RemainingBalance
FROM deleted
INNER JOIN inserted
ON deleted.CampaignID = inserted.CampaignID
WHERE ABS(deleted.RemainingBalance - inserted.RemainingBalance) > 10000;
END;
GO
```

6. In the toolbar, click **Execute**.
7. Do not close SQL Server Management Studio.

► Task 2: Delete all Rows from the Marketing.CampaignAudit Table

1. Click **New Query**.
2. In the query pane, type the following query:

```
DELETE FROM Marketing.CampaignAudit;
GO
```

3. In the toolbar, click **Execute**.
4. Do not close SQL Server Management Studio.

► Task 3: Test the Modified Trigger

1. Click **New Query**.
2. In the query pane, type the following query:

```
SELECT * FROM Marketing.CampaignBalance;
GO
EXEC Marketing.MoveCampaignBalance 3,2,10100;
GO
EXEC Marketing.MoveCampaignBalance 3,2,1010;
GO
SELECT * FROM Marketing.CampaignAudit;
GO
```

3. In the toolbar, click **Execute**.
4. Close SQL Server Management Studio without saving anything.

Results: After this exercise, you should have altered the trigger. Tests should show that it is now working as expected.

Module 12: Using In-Memory Tables

Lab: Using In-Memory Database Capabilities

Exercise 1: Using Memory-Optimized Tables

► **Task 1: Prepare the Lab Environment**

1. Ensure that the MIA-DC and MIA-SQL virtual machines are both running, and then log on to MIA-SQL as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab12\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes** to confirm that you want to run the command file, and then wait for the script to finish.

► **Task 2: Add a Filegroup for Memory-Optimized Data**

1. Start SQL Server Management Studio, and then connect to the **MIA-SQL** database engine instance by using Windows authentication.
2. In SQL Server Management Studio, in Object Explorer, expand **Databases**.
3. Right-click the **InternetSales** database, and then click **Properties**.
4. In the **Database Properties - InternetSales** dialog box, on the **Filegroups** page, in the **MEMORY OPTIMIZED DATA** section, click **Add Filegroup**.
5. In the **Name** box, type **MemFG**, and then press Enter.
6. In the **Database Properties - InternetSales** dialog box, on the **Files** page, click **Add**.
7. In the **Logical Name** column, type **InternetSales_MemData**, and then press Enter.
8. In the **File Type** column, select **FILESTREAM Data**, and then ensure that **MemFG** is automatically selected in the **Filegroup** column.
9. In the **Database Properties - InternetSales** dialog box, click **OK**.

► **Task 3: Create a Memory-Optimized Table**

1. In SQL Server Management Studio, click **New Query**, and then type the following Transact-SQL code.

```
USE InternetSales
GO
CREATE TABLE dbo.ShoppingCart
(SessionID INT NOT NULL,
TimeAdded DATETIME NOT NULL,
CustomerKey INT NOT NULL,
ProductKey INT NOT NULL,
Quantity INT NOT NULL
PRIMARY KEY NONCLUSTERED (SessionID, ProductKey))
WITH (MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA);
```

2. On the toolbar, click **Execute** to create the table.

3. Click **New Query**, and then type the following Transact-SQL code.

```
USE InternetSales
GO
INSERT INTO dbo.ShoppingCart (SessionID, TimeAdded, CustomerKey, ProductKey,
Quantity)
VALUES (1, GETDATE(), 2, 3, 1);
INSERT INTO dbo.ShoppingCart (SessionID, TimeAdded, CustomerKey, ProductKey,
Quantity)
VALUES (1, GETDATE(), 2, 4, 1);
SELECT * FROM dbo.ShoppingCart;
```

4. On the toolbar, click **Execute** to test the table.

Results: After completing this exercise, you should have created a memory-optimized table and a natively compiled stored procedure in a database with a filegroup for memory-optimized data.

Exercise 2: Using Natively Compiled Stored Procedures

► Task 1: Create Natively Compiled Stored Procedures

1. In SQL Server Management Studio, click **New Query**, and then enter the following Transact-SQL code.

```
USE InternetSales
GO
CREATE PROCEDURE dbo.AddItemToCart
@SessionID INT, @TimeAdded DATETIME, @CustomerKey INT, @ProductKey INT, @Quantity INT
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS
BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = 'us_english')
INSERT INTO dbo.ShoppingCart (SessionID, TimeAdded, CustomerKey, ProductKey,
Quantity)
VALUES (@SessionID, @TimeAdded, @CustomerKey, @ProductKey, @Quantity)
END
GO
```

2. On the toolbar, click **Execute** to create the stored procedure.
3. In SQL Server Management Studio, click **New Query**, and then enter the following Transact-SQL code.

```
USE InternetSales
GO
CREATE PROCEDURE dbo.DeleteItemFromCart
@SessionID INT, @ProductKey INT
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS
BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = 'us_english')
DELETE FROM dbo.ShoppingCart
WHERE SessionID = @SessionID
AND ProductKey = @ProductKey
END
GO
```

4. On the toolbar, click **Execute** to create the stored procedure.

5. In SQL Server Management Studio, click **New Query**, and then enter the following Transact-SQL code.

```
USE InternetSales
GO
CREATE PROCEDURE dbo.EmptyCart
@SessionID INT
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS
BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = 'us_english')
DELETE FROM dbo.ShoppingCart
WHERE SessionID = @SessionID
END
GO
```

6. On the toolbar, click **Execute** to create the stored procedure.
 7. Click **New Query**, and then enter the following Transact-SQL code.

```
--Step 1 - Use the InternetSales database
USE InternetSales
GO
--Step 2 - Add items to cart
DECLARE @now DATETIME = GETDATE();
EXEC dbo.AddItemToCart
  @SessionID = 3,
  @TimeAdded = @now,
  @CustomerKey = 2,
  @ProductKey = 3,
  @Quantity = 1;
EXEC dbo.AddItemToCart
  @SessionID = 3,
  @TimeAdded = @now,
  @CustomerKey = 2,
  @ProductKey = 4,
  @Quantity = 1;
--Step 3 - Select items in cart
SELECT * FROM dbo.ShoppingCart;
--Step 4 - Delete item from cart
EXEC dbo.DeleteItemFromCart
  @SessionID = 3,
  @ProductKey = 4;
--Step 5 - Select items in cart
SELECT * FROM dbo.ShoppingCart;
--Step 6 - Empty cart
EXEC dbo.EmptyCart
  @SessionID = 3;
--Step 7 - Select items in cart
SELECT * FROM dbo.ShoppingCart;
```

8. Select the code under **Step 1 - Use the InternetSales database**, and then click **Execute**.
 9. Select the code under **Step 2 - Add items to cart**, and then click **Execute**.
 10. Select the code under **Step 3 - Select items in cart**, and then click **Execute**.
 11. Select the code under **Step 4 - Delete item from cart**, and then click **Execute**.
 12. Select the code under **Step 5 - Select items in cart**, and then click **Execute**.
 13. Select the code under **Step 6 - Empty cart**, and then click **Execute**.
 14. Select the code under **Step 7 - Select items in cart**, and then click **Execute**.
 15. Close SQL Server Management Studio without saving any changes.

Results: After completing this exercise, you should have created a natively compiled stored procedure.

Module 13: Implementing Managed Code in SQL Server

Lab: Implementing Managed Code in SQL Server

Exercise 1: Assessing Proposed CLR Code

► Task 1: Prepare the Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab13\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

► Task 2: Review the List of Proposed Functionality

1. In File Explorer, browse to the **D:\Labfiles\Lab13\Starter** folder, and then double-click **TSQL_or_Managed_Code.docx**.
2. Review the proposed functionality and, for each row, enter your recommended choice of implementation (SQL CLR or Transact-SQL) in column B, and your reasons for that choice in column C.
3. On the **File** menu, click **Save**.
4. On the **File** menu, click **Open**.
5. In the **Open** pane, click **Browse**.
6. In the **Open** dialog box, browse to the **D:\Labfiles\Lab13\Solution** folder, click **TSQL_or_Managed_Code_Solution.docx**, and then click **Open**.
7. Compare your choices with the answers in the solution file.
8. Close any open WordPad windows when you have finished.

Results: After completing this lab, you will have determined which type of code to use for each new feature.

Exercise 2: Creating a Scalar-Valued CLR Function

► Task 1: Create a Scalar-Valued Function

1. Start SQL Server Management Studio, and then connect to the **MIA-SQL** instance by using Windows authentication.
2. On the taskbar, click **Visual Studio 2015**.
3. On the **File** menu, point to **Open**, and then click **Project/Solution**.
4. In the **Open Project** dialog box, navigate to **D:\Labfiles\Lab13\Starter\ClrPractice**, click **ClrPractice.sln**, and then click **Open**.
5. In Solution Explorer, double-click **IsRegexMatch.cs**.
6. Review the code in the file.
7. On the **Build** menu, click **Build Solution**, and wait for the build to complete.

► Task 2: Publish the Scalar-Valued Function

1. In Solution Explorer, under **Solution 'ClrPractice'**, right-click **ClrPractice**, and then click **Publish**.
2. In the **Publish Database** dialog box, click **Edit**.
3. In the **Connect** dialog box, on the **Browse** tab, expand **Local**, and then click **MIA-SQL**.
4. In the **Database Name** list, click **AdventureWorks2014**, and then click **Test Connection**.
5. In the **Connect** message box, click **OK**.
6. In the **Connect** dialog box, click **OK**.
7. In the **Publish Database** dialog box, click **Publish**, and wait for the publish process to complete.

Observe that publishing fails—this is because the assembly is not signed with a strong name that is trusted by the database. Click **View Results** in the **Data Tools Operations** pane to view the detailed error message.

► Task 3: Create an Asymmetric Key and a Login in the Database

1. In SSMS, on the **File** menu, point to **Open**, and then click **File**.
2. In the **Open File** dialog box, navigate to the **D:\Labfiles\Lab13\Starter** folder, click **Create_asymmetric_key.sql**, and then click **Open**.
3. Select the code under the comment that begins **Step 1** then click **Execute** to reset the database to a known state.
4. Edit the code under the comment that begins **Step 2** so that it reads:

```
CREATE ASYMMETRIC KEY assembly_key FROM FILE =
'D:\Labfiles\Lab13\Starter\strong_name.snk';
```

select the code you have just edited, then click **Execute**.

5. Edit the code under the comment that begins **Step 3** so that it reads:

```
CREATE LOGIN sign_assemblies FROM ASYMMETRIC KEY assembly_key;
```

select the code you have just edited, then click **Execute**.

6. Select the code under the comment that begins **Step 4** then click **Execute** to grant the UNSAFE ASSEMBLY permission to the new login.

► **Task 4: Sign and Publish the Assembly**

1. In Visual Studio, in Solution Explorer, under **Solution 'ClrPractice'**, right-click **ClrPractice**, and then click **Properties**.
2. In the **ClrProperties** pane, on the **SQLCLR** page, click **Signing**.
3. In the **Signing** dialog, select **Sign the assembly**. In the **Choose a strong name key file** box, select **Browse**.
4. In the **Select file** dialog, in the **File name** box type **D:\Labfiles\Lab13\Starter\strong_name.snk** then click **Open**, then click **OK**.
5. In Solution Explorer, under **Solution 'ClrPractice'**, right-click **ClrPractice**, and then click **Rebuild**.
6. In Solution Explorer, under **Solution 'ClrPractice'**, right-click **ClrPractice**, and then click **Publish**.
7. In the **Publish Database** dialog box, click **Edit**.
8. In the **Connect** dialog box, on the **Browse** tab, expand **Local**, and then click **MIA-SQL**.
9. In the **Database Name** list, click **AdventureWorks2014**, and then click **Test Connection**.
10. In the **Connect** message box, click **OK**.
11. In the **Connect** dialog box, click **OK**.
12. In the **Publish Database** dialog box, click **Publish**, and wait for the publish process to complete.

► **Task 5: Test the Scalar-Valued Function**

1. In SSMS, in Object Explorer, expand **Databases**, expand **AdventureWorks2014**, expand **Programmability**, and then expand **Assemblies**. Verify that your new assembly **ClrPractice** is listed.
2. Under the **Programmability** node, expand **Functions**, and then expand **Scalar-valued Functions**. Verify that your new **dbo.IsRegExMatch** function is listed.
3. On the **File** menu, point to **Open**, and then click **File**.
4. In the **Open File** dialog box, navigate to the **D:\Labfiles\Lab13\Starter** folder, click **RegExMatch.sql**, and then click **Open**.
5. Review the code, and then on the toolbar, click **Execute**. Verify that the query successfully uses the **IsRegExMatch** function to return **1** if the product name contains the word wheel.
6. On the **File** menu, click **Close**.
7. Keep SSMS and Visual Studio open for the next exercise.

Results: After completing this exercise, you will have a scalar-valued CLR function available in SQL Server Management Studio.

Exercise 3: Creating a Table-Valued CLR Function

► Task 1: Create a Table-Valued function

1. In Visual Studio, in Solution Explorer, double-click **RegexMatchesTV.cs**.
2. Review the code in the file.
3. On the **Build** menu, click **Build Solution**, and wait for the build to complete.

► Task 2: Publish and Test the Table-Valued Function

1. In Solution Explorer, under **Solution 'ClrPractice'**, right-click **ClrPractice**, and then click **Publish**.
2. In the **Publish Database** dialog box, click **Edit**.
3. In the **Connect** dialog box, on the **History** tab, click **AdventureWorks2014**, and then click **Test Connection**.
4. In the **Connect** message box, click **OK**.
5. In the **Connect** dialog box, click **OK**.
6. In the **Publish Database** dialog box, click **Publish**, and wait for the publish process to complete.
7. In SSMS, in Object Explorer, under **Functions**, expand **Table-valued Functions**. Verify that your new **dbo.RegexMatches** function is listed.
8. On the **File** menu, point to **Open**, and then click **File**.
9. In the **Open File** dialog box, navigate to the **D:\Labfiles\Lab13\Starter** folder, click **Test_RegExMatches.sql**, and then click **Open**.
10. Review the code, and then on the toolbar, click **Execute**. Verify that the queries successfully use the **RegexMatches** function to return matching rows.
11. If time permits, you could test the **StringAggregate** function by using **Test_StringAggregate.sql**.
12. Close SSMS without saving any changes.
13. Close Visual Studio without saving any changes.

Results: After completing this exercise, you will have a table-valued CLR function available in SQL Server Management Studio.

Module 14: Storing and Querying XML Data in SQL Server

Lab: Storing and Querying XML Data in SQL Server

Exercise 1: Determining When to Use XML

► **Task 1: Prepare the Lab Environment**

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab14\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

► **Task 2: Review the List of Scenario Requirements**

1. Review the list of requirements.
2. The following table shows which requirements are suitable for XML storage:

Requirements	Should be implemented	Reason
Existing XML data that is stored, but not processed.	Yes	Data is already XML and does not need to be processed.
Storing attributes for a customer.	No	Should be database columns.
Relational data that is being passed through a system, but not processed within it.	Perhaps	Only if the data is being sent and received as XML.
Storing attributes that are nested (that is, attributes stored within attributes).	Yes	Not standard relational data.

Results: After completing this exercise, you will have determined the appropriate use cases for XML storage.

Exercise 2: Testing XML Data Storage in Variables

► Task 1: Review, Execute, and Review the Results of the XML Queries

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, and then click **Connect**.
3. On the **File** menu, point to **Open**, and then click **File**.
4. In the **Open File** dialog box, navigate to **D:\Labfiles\Lab14\Starter**, click **InvestigateStorage.sql**, and then click **Open**.
5. Select the following code, and then on the toolbar, click **Execute**:

```
USE tempdb;
GO
```

6. Select the code under **Step 14.1**, and then click **Execute**.
7. Select the code under **Step 14.2**, and then click **Execute**. Note that one row is inserted.
8. Select the code under **Step 14.3**, and then click **Execute**. Note that one row is inserted.
9. Select the code under **Step 14.4**, and then click **Execute**. Note that one row is inserted.
10. Select the code under **Step 14.5**, and then click **Execute**. Note that one row is inserted.
11. Select the code under **Step 14.6**, and then click **Execute**. Note that one row is inserted.
12. Select the code under **Step 14.7**, and then click **Execute**. Note that one row is inserted.
13. Select the code under **Step 14.8**, and then click **Execute**. Note that the command fails due to incorrect XML format.
14. Select the code under **Step 14.9**, and then click **Execute**.
15. On the **File** menu, click **Close**.
16. Leave SSMS open for the next exercise.

Results: After this exercise, you will have seen how XML data is stored in variables.

Exercise 3: Using XML Schemas

► Task 1: Review, Execute, and Review the Results of the XML Queries

1. In SQL Server Management Studio, on the **File** menu, point to **Open**, and then click **File**.
2. In the **Open File** dialog box, navigate to **D:\Labfiles\Lab14\Starter**, click **XMLSchema.sql**, and then click **Open**.
3. Select the following code, and then on the toolbar, click **Execute**:

```
USE tempdb;
GO
```

4. Select the code under **Step 14.10**, and then click **Execute**.

5. Select the code under **Step 14.11**, and then click **Execute**. Note there are ten rows showing how the individual components of the XML schema collection are stored. Note the values in the **kind_desc** column.
6. On the **File** menu, click **Close**.
7. Leave SSMS open for the next exercise.

Results: After this exercise, you will have seen how to create XML schema collections.

Exercise 4: Using FOR XML Queries

► Task 1: Review, Execute, and Review the Results of the FOR XML Queries

1. In SQL Server Management Studio, on the **File** menu, point to **Open**, and then click **File**.
2. In the **Open File** dialog box, navigate to **D:\Labfiles\Lab14\Starter**, click **XMLQuery.sql**, and then click **Open**.
3. Select the code under **Step 14.21**, and then click **Execute**. Note the element name is based on the table name.
4. Select the code under **Step 14.22**, and then click **Execute**. Note the element-centric output and the element name is "row".
5. Select the code under **Step 14.23**, and then click **Execute**. Note the products without color show `xsi:nil="true"` when you view the data by clicking the hyperlink. Other products show the color.
6. Select the code under **Step 14.24**, and then click **Execute**. Note the element name is now Product, based on the alias name of the table.
7. Select the code under **Step 14.25**, and then click **Execute**. Note the inclusion of an XSD schema.
8. Select the code under **Step 14.26**, and then click **Execute**. Note that rows with a value in the **Description** column show that value as XML.
9. Select the code under **Step 14.27**, and then click **Execute**. Note how the output can be constructed with a PATH query. In the output, locate each of the elements in the SELECT clause.
10. Select the code under **Step 14.28**, and then click **Execute**. Note the "AvailableItems" root node.
11. Select the code under **Step 14.29**, and then click **Execute**. Note the "AvailableItem" ElementName.
12. On the **File** menu, click **Close**.
13. Leave SSMS open for the next exercise.

Results: After this exercise, you will have seen how to use FOR XML.

Exercise 5: Creating a Stored Procedure to Return XML

► Task 1: Review the Requirements

- Review the stored procedure specification for **WebStock.GetAvailableModelsAsXML** in the exercise scenario.

► Task 2: Create a Stored Procedure to Retrieve Available Models

1. In Object Explorer, expand **Databases**, right-click **AdventureWorks**, and then click **New Query**.
2. In the query pane, type the following query, and then click **Execute**:

```
CREATE PROCEDURE
Production.GetAvailableModelsAsXML
AS BEGIN
SELECT p.ProductID,
p.name as ProductName,
p.ListPrice,
p.Color,
p.SellStartDate,
pm.ProductModelID,
pm.Name as ProductModel
FROM Production.Product AS p
INNER JOIN Production.ProductModel AS pm
ON p.ProductModelID = pm.ProductModelID
WHERE p.SellStartDate IS NOT NULL
AND p.SellEndDate IS NULL
ORDER BY p.SellStartDate, p.Name DESC
FOR XML RAW('AvailableModel'), ROOT('AvailableModels');
END;
GO
```

► Task 3: Test the Stored Procedure

1. In Object Explorer, right-click **AdventureWorks**, and then click **New Query**.
2. In the query pane, type the following query, and then click **Execute**:

```
EXEC Production.GetAvailableModelsAsXML;
GO
```

3. In the results pane, click the XML code in the first row.
4. Review the data that the stored procedure returns.

► **Task 4: If Time Permits: Create a Stored Procedure to Update the Sales Territories Table**

1. In Object Explorer, right-click **AdventureWorks**, and then click **New Query**.
2. In the query pane, type the following query, and then click **Execute**:

```
CREATE PROCEDURE Sales.UpdateSalesTerritoriesByXML (@SalespersonMods as xml)
AS BEGIN
    UPDATE Sales.SalesPerson
    SET TerritoryID = updates.SalesTerritoryID
    FROM Sales.SalesPerson sp
    INNER JOIN (
        SELECT
            SalespersonMod.value('@BusinessEntityID','int') AS BusinessEntityID,
            SalespersonMod.value('(Mods/Mod/@SalesTerritoryID)[1]','int') AS SalesTerritoryID
        FROM
            @SalespersonMods.nodes('/SalespersonMods/SalespersonMod') as
            SalespersonMods(SalespersonMod) AS updates
        ON sp.BusinessEntityID = updates.BusinessEntityID;
    END;
    GO
```

3. In Object Explorer, right-click **AdventureWorks**, and then click **New Query**.
4. In the query pane, type the following query to test your stored procedure, and then click **Execute**:

```
DECLARE @xmlTestString nvarchar(2000);
SET @xmlTestString =
<SalespersonMods>
    <SalespersonMod BusinessEntityID="274">
        <Mods>
            <Mod SalesTerritoryID="3"/>
        </Mods>
    </SalespersonMod>
    <SalespersonMod BusinessEntityID="278">
        <Mods>
            <Mod SalesTerritoryID="4"/>
        </Mods>
    </SalespersonMod>
</SalespersonMods>
)';
DECLARE @testDoc xml;
SET @testDoc = @xmlTestString;
EXEC Sales.UpdateSalesTerritoriesByXML @testDoc;
GO
```

5. Note that two rows in the database have been updated.
6. Close SSMS without saving any changes.

Results: After this exercise, you will have a new stored procedure that returns XML in the AdventureWorks database.

Module 15: Storing and Querying Spatial Data in SQL Server

Lab: Working with SQL Server Spatial Data

Exercise 1: Become Familiar with the geometry Data Type

► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In File Explorer, navigate to the **D:\Labfiles\Lab15\Starter** folder, right-click the **Setup.cmd** file, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish. Press any key to continue.

► Task 2: Review and Execute Queries

1. Start SQL Server Manager Studio and connect to the **MIA-SQL** database instance using Windows authentication.
2. On the **File** menu, click **Open**, click **Project/Solution**.
3. In the **Open Project** dialog box, navigate to **D:\Labfiles\Lab15\Starter**, and then double-click **20762_15.ssmssln**.
4. In Solution Explorer, expand **Queries**, and double-click **51 - Lab Exercise 1.sql** to open the query.
5. Highlight the text under the comment **Script 15.1 Draw a square**, and click **Execute**.
6. Click the **Spatial results** tab to see the output of the script.
7. Highlight the text under the comment **Script 15.2 Try an invalid value - note the 6522 error and the wrapped 24306 error message**, and click **Execute**.
8. Review the error message produced.
9. Highlight the text under the comment **Script 15.3 Draw a more complex shape**, and click **Execute**.
10. Click the **Spatial results** tab to see the output of the script.
11. Highlight the text under the comment **Script 15.4 Multiple shapes**, and click **Execute**.
12. Click the **Spatial results** tab to see the output of the script.
13. Highlight the text under the comment **Script 15.5 Intersecting shapes**, and click **Execute**.
14. Click the **Spatial results** tab to see the output of the script.
15. Highlight the text under the comment **Script 15.6 Union of the two shapes**, and click **Execute**.
16. Click the **Spatial results** tab to see the output of the script.
17. Highlight the text under the comment **Script 15.7 Intersection of shapes**, and click **Execute**.
18. Click the **Spatial results** tab to see the output of the script.
19. Highlight the text under the comment **Script 15.8 Draw Australia**, and click **Execute**.
20. Click the **Spatial results** tab to see the output of the script.
21. Highlight the text under the comment **Script 15.9 Draw Australia with a buffer around it**, and click **Execute**.
22. Click the **Spatial results** tab to see the output of the script.

Exercise 2: Add Spatial Data to an Existing Table

► Task 1: Add a Location Column to the Marketing.ProspectLocation Table

1. In Object Explorer, expand **MIA-SQL (SQL Server 13.0.1000 - ADVENTUREWORKS\Student)**, expand **Databases**, right-click the **MarketDev** database, and then click **New Query**.
2. In the query pane, type the following query:

```
ALTER TABLE Marketing.ProspectLocation
ADD Location GEOGRAPHY NULL;
GO
```

3. On the toolbar, click **Execute**.

► Task 2: Write Code to Assign Values Based on Existing Latitude and Longitude Columns

1. In Object Explorer, right-click the **MarketDev** database, and then click **New Query**.
2. In the query pane, type the following query:

```
UPDATE Marketing.ProspectLocation
SET Location = GEOGRAPHY::STGeomFromText('POINT(' + CAST(Longitude AS varchar(20))+ '
' + CAST(Latitude AS varchar(20))+ ')',4326);
GO
```

3. On the toolbar, click **Execute**.

► Task 3: Drop the Existing Latitude and Longitude Columns

1. In Object Explorer, right-click the **MarketDev** database, and then click **New Query**.
2. In the query pane, type the following query:

```
ALTER TABLE Marketing.ProspectLocation
DROP COLUMN Latitude;
GO
ALTER TABLE Marketing.ProspectLocation
DROP COLUMN Longitude;
GO
```

3. On the toolbar, click **Execute**.

Results: After this exercise, you should have replaced the existing **Longitude** and **Latitude** columns with a new **Location** column.

Exercise 3: Find Nearby Locations

► Task 1: Review the Requirements

1. In File Explorer, navigate to the folder **D:\Labfiles\Lab15\Starter** and open the file **Requirements.docx**.
2. Read the requirements document to familiarize yourself with the requirements.

► Task 2: Create a Spatial Index on the Marketing.ProspectLocation Table

1. In SQL Server Management Studio, in Object Explorer, right-click the **MarketDev** database, and then click **New Query**.
2. In the query pane, type the following Transact-SQL, and click **Execute**:

```
CREATE SPATIAL INDEX IX_ProspectLocation_Location
ON Marketing.ProspectLocation(Location);
GO
```

► Task 3: Design and Implement the Stored Procedure

1. In Object Explorer, right-click the **MarketDev** database, and then click **New Query**.
2. In the query pane, type the following Transact-SQL, and click **Execute**:

```
CREATE PROCEDURE Marketing.GetNearbyProspects
( @ProspectID int,
  @DistanceInKms int
)
AS BEGIN
    DECLARE @LocationToTest GEOGRAPHY;
    SET @LocationToTest = (SELECT p1.Location
                           FROM Marketing.ProspectLocation AS p1
                           WHERE p1.ProspectID = @ProspectID);
    SELECT p1.Location.STDistance(@LocationToTest) / 1000 AS Distance,
          p.ProspectID,
          p.LastName,
          p.FirstName,
          p.WorkPhoneNumber,
          p.CellPhoneNumber,
          p1.AddressLine1,
          p1.AddressLine2,
          p1.City
    FROM Marketing.Prospect AS p
    INNER JOIN Marketing.ProspectLocation AS p1
    ON p.ProspectID = p1.ProspectID
    WHERE p1.Location.STDistance(@LocationToTest) < (@DistanceInKms * 1000)
      AND p.ProspectID <> @ProspectID
    ORDER BY Distance;
END;
```

3. GO

► Task 4: Test the Procedure

1. In Object Explorer, right-click the **MarketDev** database, and then click **New Query**.
2. In the query pane, type the following Transact-SQL, and click **Execute**:

```
EXEC Marketing.GetNearbyProspects 2,50;
GO
```

3. Close SSMS without saving anything.

Results: After completing this lab, you will have created a spatial index and written a stored procedure that will return the prospects within a given distance from a chosen prospect.

Module 16: Storing and Querying BLOBs and Text Documents in SQL Server

Lab: Storing and Querying BLOBs and Text Documents in SQL Server

Exercise 1: Enabling and Using FILESTREAM Columns

► Task 1: Prepare the Environment

Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab16\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**.
4. At the command prompt, if the **Do you want to continue this operation?** message appears, type **y**, and press Enter.
5. Wait for the script to finish, and then press Enter.

► Task 2: Enable FILESTREAM for the SQL Server Instance

1. On the Start menu, type **SQL Server 2017 Configuration Manager**, and then click **SQL Server 2017 Configuration Manager**.
2. In the **User Account Control** dialog box, click **Yes**.
3. In the hierarchy on the left, click **SQL Server Services**.
4. In the list of services, right-click **SQL Server (MSSQLSERVER)**, and then click **Properties**.
5. In the **SQL Server (MSSQLServer) Properties** dialog box, on the **FILESTREAM** tab, ensure the **Enable FILESTREAM for Transact-SQL access** check box is selected.
6. Ensure the **Enable FILESTREAM for file I/O access** check box is selected.
7. Ensure the **Allow remote clients access to FILESTREAM data** check box is selected, and then click **OK**.
8. In the list of services, right-click **SQL Server (MSSQLSERVER)**, and then click **Restart**.
9. When the service has restarted, close SQL Server Configuration Manager.
10. On the taskbar, click **Microsoft SQL Server Management Studio**.
11. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, and then click **Options**.
12. On the **Connection Properties** tab, in the **Connect to database** list, click **<Browse server...>**.
13. In the **Browse for Databases** dialog box, click **Yes**.
14. In the **Browse Server for Databases** dialog box, under **User Databases**, click **AdventureWorks2016**, and then click **OK**.

15. In the **Connect to Server** dialog box, on the **Login** tab, in the **Authentication** list, click **Windows Authentication**, and then click **Connect**.
16. On the **File** menu, point to **Open** and click **Project/Solution**.
17. In the **Open Project** dialog box, navigate to **D:\Labfiles\Lab16\Starter\Project**, click **Project.ssmssqln**, and then click **Open**.
18. In Solution Explorer, expand **Queries**, and double-click the **Lab Exercise 1.sql** query.
19. In the query pane, after the **Task 1** description, type the following script:

```
EXEC sp_configure  
    filestream_access_level, 2;  
RECONFIGURE;  
GO
```

20. Select the script and click **Execute**.

► Task 3: Enable FILESTREAM for the Database

1. In SQL Server Management Studio, in the query pane, after the **Task 2** description, type the following script:

```
ALTER DATABASE AdventureWorks2016  
ADD FILEGROUP AdworksFilestreamGroup CONTAINS FILESTREAM;  
GO
```

2. Select the query and click **Execute**.
3. Under the previously entered code, type the following script:

```
ALTER DATABASE AdventureWorks2016  
ADD FILE (NAME='FilestreamData', FILENAME='D:\FilestreamData')  
TO FILEGROUP AdworksFilestreamGroup;  
GO
```

4. Select the query and click **Execute**.

► Task 4: Create a FILESTREAM Column

1. In SQL Server Management Studio, in the query pane, after the **Task 3** description, type the following script:

```
USE AdventureWorks2016;  
GO  
ALTER TABLE Production.ProductPhoto  
ADD PhotoGuid UNIQUEIDENTIFIER NOT NULL ROWGUIDCOL UNIQUE DEFAULT newid();  
GO
```

2. Select the query and click **Execute**.
3. Under the previously entered code, type the following script:

```
ALTER TABLE Production.ProductPhoto  
SET (filestream_on = AdworksFilestreamGroup);  
GO
```

4. Select the query and click **Execute**.

5. Under the previously entered code, type the following script:

```
ALTER TABLE Production.ProductPhoto  
ADD NewLargePhoto varbinary(max) FILESTREAM NULL;  
GO
```

6. Select the query and click **Execute**.

► Task 5: Move Data into the FILESTREAM Column

1. In SQL Server Management Studio, in the query pane, after the **Task 4** description, type the following script:

```
UPDATE Production.ProductPhoto  
SET NewLargePhoto = LargePhoto;  
GO
```

2. Select the query and click **Execute**.
3. Under the previously entered code, type the following script:

```
ALTER TABLE Production.ProductPhoto  
DROP COLUMN LargePhoto;  
GO
```

4. Select the query and click **Execute**.
5. Under the previously entered code, type the following script:


```
EXEC sp_rename 'Production.ProductPhoto.NewLargePhoto', 'LargePhoto', 'COLUMN';  
GO
```

6. Select the query and click **Execute**. Note the caution.
7. On the **File** menu, click **Close**.
8. In the **Microsoft SQL Server Management Studio** dialog box, click **Yes**.

Results: At the end of this exercise, you will have:

Enabled FILESTREAM on the SQL Server instance.

Enabled FILESTREAM on a database.

Moved data into the FILESTREAM column.

Exercise 2: Enabling and Using FileTables

► Task 1: Enable Nontransactional Access

1. In SQL Server Management Studio, in Solution Explorer, double-click **Lab Exercise 2.sql**.
2. In the query pane, after the **Task 1** description, type the following script:

```
SELECT DB_NAME(database_id) AS dbname, non_transacted_access,
       non_transacted_access_desc
      FROM sys.database_filestream_options;
GO
```

3. Select the query and click **Execute**.
4. Under the previously entered code, type the following script:

```
ALTER DATABASE AdventureWorks2016
SET FILESTREAM ( NON_TRANSACTED_ACCESS = FULL, DIRECTORY_NAME =
N'FileTablesDirectory' );
GO
```

5. Select the query and click **Execute**.

► Task 2: Create a FileTable

1. In SQL Server Management Studio, in the query pane, after the **Task 2** description, type the following script:

```
USE AdventureWorks2016;
GO
CREATE TABLE dbo.DocumentStore AS FileTable
WITH (FileTable_Directory = 'DocumentStore');
GO
```

2. Select the query and click **Execute**.
3. Under the previously entered code, type the following script:

```
SELECT DB_NAME(database_id) AS dbname, non_transacted_access,
       non_transacted_access_desc
      FROM sys.database_filestream_options;
GO
```

4. Select the query and click **Execute**.
5. Under the previously entered code, type the following script:

```
SELECT * FROM AdventureWorks2016.sys.filetables;
```

6. Select the query and click **Execute**.

► Task 3: Add a File to the FileTable

1. In SQL Server Management Studio, in the query pane, after the **Task 3** description, type the following script:

```
SELECT
FileTableRootPath('dbo.DocumentStore');
GO
```

2. Select the query and click **Execute**.

3. In the **Results** pane, right-click the only result, and then click **Copy**.
4. On the taskbar, right-click **File Explorer**, and then click **File Explorer**.
5. Click the address bar, right-click the address bar, click **Paste**, and then press Enter.
6. In the **DocumentStore** folder, on the **Home** menu, click **New item**, and then click **Text Document**.
7. Type **DocumentStoreTest**, and then press Enter.
8. Switch to SQL Server Management Studio.
9. Under the previously entered code, type the following script:

```
SELECT * FROM dbo.DocumentStore;
```

10. Select the query and click **Execute**.

Results: At the end of this exercise, you will have:

Enabled nontransactional access.

Created a FileTable.

Added a file to the FileTable.

Exercise 3: Using a Full-Text Index

► Task 1: Create a Full-Text Index

1. In SQL Server Management Studio, in the Solution Explorer, double-click **Lab Exercise 3.sql**.
2. In the query pane, after the **Task 1** description, highlight and execute the script.
3. Under the previously entered code, type the following script:

```
CREATE FULLTEXT CATALOG ProductFullTextCatalog;
GO
```

4. Select the query and click **Execute**.
5. Under the previously entered code, type the following script:

```
CREATE UNIQUE INDEX ui_ProductDescriptionID ON
Production.ProductDescription(ProductDescriptionID);
GO
```

6. Select the query and click **Execute**.
7. Under the previously entered code, type the following script:

```
CREATE FULLTEXT INDEX ON Production.ProductDescription
( Description Language 1033 )
KEY INDEX ui_ProductDescriptionID
ON ProductFullTextCatalog;
GO
```

8. Select the query and click **Execute**.

► **Task 2: Using a Full-Text Index**

1. In SQL Server Management Studio, in the query pane, after the **Task 2** description, type the following script:

```
SELECT ProductDescriptionID, Description  
FROM Production.ProductDescription  
WHERE CONTAINS>Description, 'Bike');
```

2. Select the query and click **Execute**. Make a note of the number of rows returned.
3. Under the previously entered code, type the following script:

```
SELECT ProductDescriptionID, Description  
FROM Production.ProductDescription  
WHERE CONTAINS>Description, 'FORMSOF(INFLECTIONAL, Bike)');
```

4. Select the query and click **Execute**. Make a note of the number of rows returned.
 5. Under the previously entered code, type the following script:
- ```
SELECT ProductDescriptionID, Description
FROM Production.ProductDescription
WHERE CONTAINS>Description, 'FORMSOF(INFLECTIONAL, Bike)') AND NOT
CONTAINS>Description, 'Bike');
```
6. Select the query and click **Execute**. Make a note of the number of rows returned.
  7. Close Microsoft SQL Server Management Studio, without saving any changes.

**Results:** At the end of this exercise, you will have created a full-text index.

# Module 17: SQL Server Concurrency

## Lab: Concurrency and Transactions

### Exercise 1: Implement Snapshot Isolation

► **Task 1: Prepare the Lab Environment**

1. Ensure that the **MT17B-WS2016-NAT**, **20762C-MIA-DC**, and **20762C-MIA-SQL** virtual machines are running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab17\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

► **Task 2: Clear Wait Statistics**

1. Start **SQL Server Management Studio** and connect to the **MIA-SQL** database engine using Windows authentication.
2. In SQL Server Management Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, open the project **D:\Labfiles\Lab17\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **Lab Exercise 01 - snapshot isolation.sql**.
5. To clear wait statistics, select the query under the comment that begins **Task 1**, and then click **Execute**.

► **Task 3: Run the Workload**

1. Open File Explorer, navigate to the **D:\Labfiles\Lab17\Starter** folder.
2. Right-click **start\_load\_exercise\_01.ps1**, and then click **Run with PowerShell**.
3. If a message is displayed asking you to confirm a change in execution policy, type **Y**, and then press Enter.
4. Wait for the workload to complete, and then press Enter to close the window.

► **Task 4: Capture Lock Wait Statistics**

1. In SQL Server Management Studio, in the query pane, edit the query under the comment that begins **Task 3** so that it reads:

```
SELECT wait_type, waiting_tasks_count, wait_time_ms,
max_wait_time_ms, signal_wait_time_ms
INTO #task3
FROM sys.dm_os_wait_stats
WHERE wait_type LIKE 'LCK%'
AND wait_time_ms > 0
ORDER BY wait_time_ms DESC;
```

2. Select the query you have amended and click **Execute**.

► **Task 5: Enable SNAPSHOT Isolation**

1. In SQL Server Management Studio Object Explorer, under MIA-SQL, expand **Databases**, right-click **AdventureWorks**, and then click **Properties**.
2. In the **Database Properties - AdventureWorks** dialog box, on the **Options** page, in the **Miscellaneous** section, change the value of the **Allow Snapshot Isolation** setting to **True**, and then click **OK**.

► **Task 6: Implement SNAPSHOT Isolation**

1. In Solution Explorer, double-click the query **Lab Exercise 01 - stored procedure.sql**.
2. Amend the stored procedure definition in the file so that it reads:

```
USE AdventureWorks;
GO
ALTER PROC Proseware.up_Campaign_Report
AS
 SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
 SELECT TOP 10 * FROM Sales.SalesTerritory AS T
 JOIN (
 SELECT CampaignTerritoryID,
 DATEPART(MONTH, CampaignStartDate) as start_month_number,
 DATEPART(MONTH, CampaignEndDate) as end_month_number,
 COUNT(*) AS campaign_count
 FROM Proseware.Campaign
 GROUP BY CampaignTerritoryID, DATEPART(MONTH,
 CampaignStartDate),DATEPART(MONTH, CampaignEndDate)
) AS x
 ON x.CampaignTerritoryID = T.TerritoryID
 ORDER BY campaign_count;
GO
```

3. Highlight the script and click **Execute**.

► **Task 7: Rerun the Workload**

1. In the SQL Server Management Studio query window for **Lab Exercise 01 - snapshot isolation.sql**, select the query under the comment that begins **Task 1**, and then click **Execute**.
2. In File Explorer, in the **D:\Labfiles\Lab17\Starter** folder, right-click **start\_load\_exercise\_01.ps1**, and then click **Run with PowerShell**.
3. Wait for the workload to complete.

► **Task 8: Capture New Lock Wait Statistics**

1. In SQL Server Management Studio, in the query window for **Lab Exercise 01 - snapshot isolation.sql**, amend the query under the comment that begins **Task 8** so that it reads:

```
SELECT wait_type, waiting_tasks_count, wait_time_ms,
 max_wait_time_ms, signal_wait_time_ms
 INTO #task8
 FROM sys.dm_os_wait_stats
 WHERE wait_type LIKE 'LCK%'
 AND wait_time_ms > 0
 ORDER BY wait_time_ms DESC;
```

2. Select the query you have amended and click **Execute**.

► **Task 9: Compare Overall Lock Wait Time**

1. In SQL Server Management Studio, in the query pane, select the query under the comment that begins **Task 9** and click **Execute**. Compare the total **wait\_time\_ms** you have captured between the **#task3** and **#task8** temporary tables. Note that the wait time in the **#task8** table—after SNAPSHOT isolation was implemented—is lower.
2. Close the query windows for **Lab Exercise 01 - snapshot isolation.sql** and **Lab Exercise 01 - stored procedure.sql** without saving changes, but leave SQL Server Management Studio open for the next exercise.

**Results:** After this exercise, the **AdventureWorks** database will be configured to use the SNAPSHOT isolation level.

## Exercise 2: Implement Partition Level Locking

► **Task 1: Open Activity Monitor**

1. In SQL Server Management Studio, in Object Explorer, right-click **MIA-SQL** and click **Activity Monitor**.
2. In Activity Monitor, click **Resource Waits** to expand the section.

► **Task 2: Clear Wait Statistics**

1. In Solution Explorer, double-click **Lab Exercise 02 - partition isolation.sql**.
2. Select the code under the comment that begins **Task 2**, and click **Execute**.

► **Task 3: View Lock Waits in Activity Monitor**

1. In File Explorer, in the **D:\Labfiles\Lab17\Starter** folder, right-click **start\_load\_exercise\_02.ps1**, and then click **Run with PowerShell**.
2. Wait for the workload to complete (it will take a few minutes).
3. In SQL Server Management Studio, on the **MIA-SQL - Activity Monitor** tab, in the **Resource Waits** section, note the value of **Cumulative Wait Time (sec)** for the **Lock** wait type.
4. In the PowerShell workload window, press Enter to close it.

► **Task 4: Enable Partition Level Locking**

1. In SQL Server Management Studio, in the **Lab Exercise 02 - partition isolation.sql** query, under the comment that begins **Task 5**, type:

```
USE AdventureWorks;
GO
ALTER TABLE Proseware.CampaignResponsePartitioned SET (LOCK_ESCALATION = AUTO);
GO
```

2. Select the query you have typed and click **Execute**.
3. Select the query under the comment that begins **Task 2**, then click **Execute**.

► **Task 5: Rerun the Workload**

1. In File Explorer, in the **D:\Labfiles\Lab17\Starter** folder, right-click **start\_load\_exercise\_02.ps1**, and then click **Run with PowerShell**.
2. Wait for the workload to complete (it will take a few minutes).
3. In SQL Server Management Studio, on the **MIA-SQL - Activity Monitor** tab, in the **Resource Waits** section, note the value of **Cumulative Wait Time (sec)** for the **Lock** wait type.
4. Compare this value to the value you noted earlier in the exercise; the wait time will be considerably lower after you implemented partition level locking.
5. In the PowerShell workload window, press Enter to close it.
6. Close SQL Server Management Studio without saving any changes.

**Results:** After this exercise, the **AdventureWorks** database will use partition level locking.

# Module 18: Performance and Monitoring

## Lab: Monitoring, Tracing, and Baselining

### Exercise 1: Collecting and Analyzing Data Using Extended Events

#### ► Task 1: Prepare the Lab Environment

1. Ensure that the **20762C-MIA-DC** and **20762C-MIA-SQL** virtual machines are both running, and then log on to **20762C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab18\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**.
4. If you are prompted with the question **Do you want to continue with this operation?**, type **y**, press Enter, and then wait for the script to finish.

#### ► Task 2: Set Up an Extended Event Session

1. Start SQL Server Management Studio, and then connect to the **MIA-SQL** database engine instance by using Windows authentication.
2. On the **File** menu, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab18\Starter\20762** folder, click **20762-18.ssmsslnproj**, and then click **Open**.
4. In Solution Explorer, under **Queries**, double-click **SetupExtendedEvent.sql**.
5. Examine the contents of the script, and then click **Execute**.
6. In Object Explorer, expand **Management**, expand **Extended Events**, and then expand **Sessions**.
7. Right-click **AnalyzeSQLEE**, and then click **Watch Live Data**.

#### ► Task 3: Execute Workload

1. In File Explorer, in the **D:\Labfiles\Lab18\Starter** folder, right-click **RunWorkload.cmd**, and then click **Run as administrator**.
2. In the **User Account Control** dialog box, click **Yes**.
3. After execution of the workload completes, repeat steps 1 and 2.
4. In SQL Server Management Studio, on the **Extended Events** menu, click **Stop Data Feed**.
5. In the **MIA-SQL - AnalyzeSQLEE: Live Data** query pane, right-click the **name** column heading, and then click **Choose Columns**.
6. In the **Choose Columns** dialog box, under **Available columns**, click **duration**, click **>**, click **query\_hash**, click **>**, click **statement**, click **>**, and then click **OK**.

#### ► Task 4: Analyze Collected Data

1. In the **MIA-SQL - AnalyzeSQLEE: Live Data** query, right-click the **query\_hash** column heading, and then click **Group by this Column**.
2. Right-click the **duration** column heading, point to **Calculate Aggregation**, and then click **AVG**.
3. Right-click the **duration** column heading, and then click **Sort Aggregation Descending**.

4. Expand one of the query hash rows to observe the top statements by duration.
5. In Solution Explorer, double-click **cleanup.sql**.
6. Examine the contents of the script, and then click **Execute** to remove the Extended Event.
7. Leave SQL Server Management Studio open for the next exercise.

**Results:** At the end of this lab, you will be able to:

Set up an Extended Events session that collects performance data for a workload.

Analyze the data.

## Exercise 2: Implementing Baseline Methodology

### ► Task 1: Set Up Data Collection Scripts

1. In SQL Server Management Studio, in Solution Explorer, double-click **PrepareScript.sql**.
2. Examine the contents of the script, and then click **Execute**. The error can be ignored because this means the database has already been removed.

### ► Task 2: Execute Workload

1. In Solution Explorer, double-click **WaitsCollectorJob.sql**.
2. Examine the contents of the script, and then click **Execute**.
3. In Object Explorer, expand **SQL Server Agent**, expand **Jobs**, right-click **waits\_collections**, and then click **Start Job at Step**.

 **Note:** If SQL Server Agent is not running, right-click **SQL Server Agent**, and then click **Restart**.

4. Wait for the job to complete, and then click **Close**.
5. In File Explorer, navigate to the **D:\Labfiles\Lab18\Starter** folder, right-click **RunWorkload.cmd**, and then click **Run as administrator**.
6. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.
7. In SQL Server Management Studio, in Object Explorer, under **Jobs**, right-click **waits\_collections**, and then click **Start Job at Step**.
8. Wait for the job to complete, and then click **Close**.

### ► Task 3: Analyze Data

1. In Solution Explorer, double-click **WaitBaselineDelta.sql**.
2. Examine the contents of the script, and then click **Execute**.
3. In Solution Explorer, double-click **WaitBaselinePercentage.sql**.
4. Examine the contents of the script, and then click **Execute**.
5. In Solution Explorer, double-click **WaitBaselineTop10.sql**.
6. Examine the contents of the script, and then click **Execute**.
7. In the **Results** pane, observe the top 10 waits that were collected during the execution of the workload.

8. Close SQL Server Management Studio without saving any changes.
9. Close File Explorer.

**Results:** After completing this exercise, you will have implemented a baseline for a workload.

