Microsoft

Microsoft Official Course

DP-080T00

Querying Data with Microsoft Transact-SQL

DP-080T00

Querying Data with Microsoft Transact-SQL

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

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

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

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

© 2019 Microsoft Corporation. All rights reserved.

Microsoft and the trademarks listed at **http://www.microsoft.com/trademarks** [1]are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

1 http://www.microsoft.com/trademarks

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

MICROSOFT LICENSE TERMS

MICROSOFT INSTRUCTOR-LED COURSEWARE

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

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

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

1. **DEFINITIONS.**
    1. "Authorized Learning Center" means a Microsoft Imagine Academy (MSIA) Program Member, Microsoft Learning Competency Member, or such other entity as Microsoft may designate from time to time.
    2. "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.
    3. "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.
    4. "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 an MPN Member (defined below), or (iii) a Microsoft full-time employee, a Microsoft Imagine Academy (MSIA) Program Member, or a Microsoft Learn for Educators – Validated Educator.
    5. "Licensed Content" means the content accompanying this agreement which may include the Microsoft Instructor-Led Courseware or Trainer Content.
    6. "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.
    7. "Microsoft Instructor-Led Courseware" means the Microsoft-branded instructor-led training course that educates IT professionals, developers, students at an academic institution, and other learners on Microsoft technologies. A Microsoft Instructor-Led Courseware title may be branded as MOC, Microsoft Dynamics, or Microsoft Business Group courseware.
    8. "Microsoft Imagine Academy (MSIA) Program Member" means an active member of the Microsoft Imagine Academy Program.
    9. "Microsoft Learn for Educators – Validated Educator" means an educator who has been validated through the Microsoft Learn for Educators program as an active educator at a college, university, community college, polytechnic or K-12 institution.
    10. "Microsoft Learning Competency Member" means an active member of the Microsoft Partner Network program in good standing that currently holds the Learning Competency status.
    11. "MOC" means the "Official Microsoft Learning Product" instructor-led courseware known as Microsoft Official Course that educates IT professionals, developers, students at an academic institution, and other learners on Microsoft technologies.
    12. "MPN Member" means an active Microsoft Partner Network program member in good standing.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

13. "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.
14. "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.
15. "Trainer" means (i) an academically accredited educator engaged by a Microsoft Imagine Academy Program Member to teach an Authorized Training Session, (ii) an academically accredited educator validated as a Microsoft Learn for Educators – Validated Educator, and/or (iii) a MCT.
16. "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.

1. **If you are a Microsoft Imagine Academy (MSIA) Program Member:**

1. 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.
2. 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.

3. For each license you acquire, you must comply with the following:

1. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
2. 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,
3. 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

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

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,

4. 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,
5. 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,
6. you will only deliver a maximum of 15 hours of training per week for each Authorized Training Session that uses a MOC title, and
7. you acknowledge that Trainers that are not MCTs will not have access to all of the trainer resources for the Microsoft Instructor-Led Courseware.

2. **If you are a Microsoft Learning Competency Member:**

   1. Each license acquire 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.
   2. For each license you acquire on behalf of an End User or MCT, 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) MCT with the unique redemption code and instructions on how they can access one (1) Trainer Content.
   3. For each license you acquire, you must comply with the following:
      1. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
      2. 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,
      3. 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,

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

4. you will ensure that each MCT teaching an Authorized Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Authorized Training Session,
5. 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,
6. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and
7. you will only provide access to the Trainer Content to MCTs.

3. **If you are a MPN Member:**
   1. 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.
   2. 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.
   3. For each license you acquire, you must comply with the following:
      1. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
      2. 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,
      3. 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,
      4. 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,

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

5. 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,
6. 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,
7. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and
8. you will only provide access to the Trainer Content to Trainers.

4. **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.
5. **If you are a Trainer.**
   1. 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.
   2. If you are an MCT, 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.
   3. 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 that Microsoft, not the third party, licenses to you under this agreement. Notices, if any, for the third party code 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.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

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:

    1. **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.

    2. **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.

    3. **Pre-release Term.** If you are an Microsoft Imagine Academy Program Member, Microsoft Learning Competency Member, MPN Member, Microsoft Learn for Educators – Validated Educator, 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

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

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 provided "as is", we are not obligated to 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.**
    1. 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.
    2. 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.**

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

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 contre-façon 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 April 2019

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Contents

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Module 0 Introduction

Course Introduction

Course Introduction

Welcome to DP-080!

This course will teach you how to use Transact-SQL to query and modify data in relational databases that are hosted in Microsoft SQL Server-based database systems, including:

- Microsoft SQL Server
- Azure SQL Database
- Azure Synapse Analytics

While the specific Transact-SQL statements supported by these different systems can vary, the majority of techniques described in this course can be used across all of them. Additionally, because Transact-SQL is based on the ANSI standard SQL language, the techniques learned on this course can often be used, with minimal adaptation, on other popular database systems.

The ability to query relational data sources is a fundamental skill for anyone working professionally with data; whether it be as a database administrator, a data analyst, or a software engineer building applications that store and consume data.

This course has been designed to focus on the kinds of queries that are most commonly used. Transact-SQL is a comprehensive language that includes many advanced capabilities not covered in this course. However, you should learn enough to create the queries you need for the most common scenarios, including:

- Using SELECT to retrieve columns from a table
- Sorting and filtering query results
- Using joins and subqueries to retrieve data from multiple tables
- Using built-in functions, aggregations, and groupings
- Inserting, updating, and deleting data

You can also learn more about Transact-SQL in the **Transact-SQL language reference**[1] in the official documentation. There is also more training available on **Microsoft Learn**[2].

Course Agenda

This course is designed to be delivered in two days, and includes the following modules:

- Module 1: Getting Started with Transact-SQL
- Module 2: Sorting and Filtering Query Results
- Module 3: Using Joins and Subqueries
- Module 4: Using Built-in Functions
- Module 5: Modifying Data

Lab Environment

This course includes hands-on activities to reinforce the concepts taught and provide a practical learning experience.

To complete the labs, you will require:

- The sample database used in the labs, hosted in a SQL Server based database.
- Azure Data Studio (or another Transact-SQL query tool of your choice).

When taking this course through an authorized learning partner, the development environment is provided as a cloud-hosted virtual machine, which includes an instance of SQL Server Express with the required database, and Azure Data Studio.

Should you wish to complete the labs using your own environment and Azure subscription, you can do so. You'll find the lab instructions, along with guidance for setting up a development environment at **https://microsoftlearning.github.io/dp-080-Transact-SQL/**.

1 https://docs.microsoft.com/sql/t-sql/language-reference
2 https://docs.microsoft.com/learn/

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Module 1 Getting Started with Transact-SQL

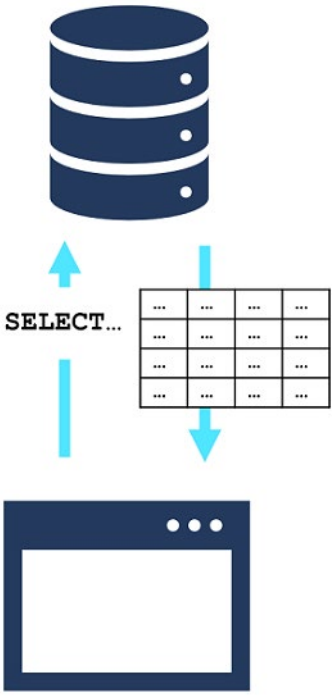
Introduction to Transact-SQL

What is Transact-SQL?

SQL is an acronym for Structured Query Language. SQL is used to communicate with relational databases. SQL statements are used to perform tasks such as update data in a database, or retrieve data from a database. For example, the SQL **SELECT** statement is used to query the database and return a set of data rows. Some common relational database management systems that use SQL include Microsoft SQL Server, MySQL, PostgreSQL, MariaDB, and Oracle.

There is a SQL language standard defined by the American National Standards Institute (ANSI). Each vendor adds their own variations and extensions.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED



Transact-SQL

Basic SQL statements, such as **SELECT**, **INSERT**, **UPDATE**, and **DELETE** are available no matter what relational database system you're working with. Although these SQL statements are part of the ANSI SQL standard, many database management systems also have their own extensions. These extensions provide functionality not covered by the SQL standard, and include areas such as security management and programmability. Microsoft database systems such as SQL Server, Azure SQL Database, Azure Synapse Analytics, and others use a dialect of SQL called Transact-SQL, or *T-SQL*. T-SQL includes language extensions for writing stored procedures and functions, which are application code that is stored in the database, and managing user accounts.

SQL is *Declarative*

Programming languages can be categorized as *procedural* or *declarative*. Procedural languages enable you to define a sequence of instructions that the computer follows to perform a task. Declarative languages enable you to describe the output you want, and leave the details of the steps required to produce the output to the execution engine.

SQL supports some procedural syntax, but querying data with SQL usually follows declarative semantics. You use SQL to describe the results you want, and the database engine's query processor develops a *query plan* to retrieve it. The query processor uses statistics about the data in the database and indexes that are defined on the tables to come up with a good query plan.

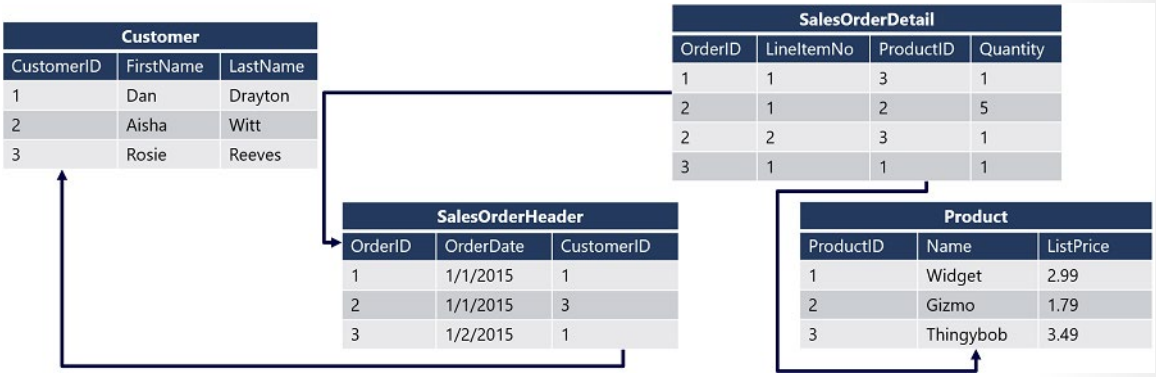
Relational Data

SQL is most often (though not always) used to query data in *relational* databases. A relational database in one in which the data has been organized in multiple tables (technically referred to as *relations*), each

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

representing a particular type of entity (such as a customer, product, or sales order). The attributes of these entities (for example, a customer's name, a product's price, or a sales order's order date) are defined as the columns, or attributes, of the table, and each row in the table represents an instance of the entity type (for example, a specific customer, product, or sales order).

The tables in the database are related to one another using *key* columns that uniquely identify the particular entity represented. A *primary key* is defined for each table, and a reference to this key is defined as a *foreign key* in any related table. This is easier to understand by looking at an example:



The diagram shows a relational database that contains four tables:

- **Customer**
- **SalesOrderHeader**
- **SalesOrderDetail**
- **Product**

Each customer is identified by a unique *CustomerID* field - this field is the primary key for the **Customer** table. The **SalesOrderHeader** table has a primary key named **OrderID** to identify each order, and it also includes a **CustomerID** foreign key that references the primary key in the **Customer** table so identify which customer is associated with each order. Data about the individual items in an order are stored in the **SalesOrderDetail** table, which has a *composite* primary key that combines the **OrderID** in the **SalesOrderHeader** table with a **LineItemNo** value. The combination of these values uniquely identifies a line item. The **OrderID** field is also used as a foreign key to indicate which order the line item belongs to, a **ProductID** field is used as a foreign key to the **ProductID** primary key of the **Product** table to indicate which product was ordered.

Set-based processing

Set theory is one of the mathematical foundations of the relational model of data management and is fundamental to working with relational databases. While you might be able to write queries in T-SQL without a thorough understanding of sets, you may eventually have difficulty writing some of the more complex types of statements that may be needed for optimum performance.

Without diving into the mathematics of set theory, you can think of a set as "a collection of definite, distinct objects considered as a whole." In terms applied to SQL Server databases, you can think of a set as a collection of distinct objects containing zero or more members of the same type. For example, the **Customer** table represents a set: specifically, the set of all customers. You will see that the results of a SELECT statement also form a set.

As you learn more about T-SQL query statements, it is important to always think of the entire set, instead of individual members. This mindset will better equip you to write set-based code, instead of thinking

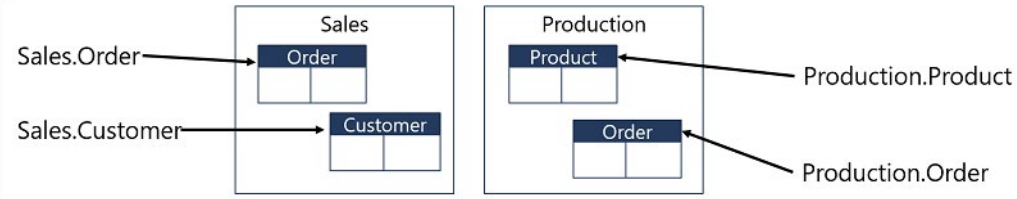
AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

one row at a time. Working with sets requires thinking in terms of operations that occur "all at once" instead of one at a time.

One important feature to note about set theory is that there is no specification regarding any ordering of the members of a set. This lack of order applies to relational database tables. There is no concept of a *first* row, a *second* row, or a *last* row. Elements may be accessed (and retrieved) in any order. If you need to return results in a certain order, you must specify it explicitly by using an ORDER BY clause in your SELECT query.

Schemas

In SQL Server database systems, tables are defined within *schemas* to create logical namespaces in the database. For example, a **Customer** table might be defined in a **Sales** schema, while a **Product** table is defined in a **Production** schema. The database might track details of orders that customers have placed in an **Order** table in the **Sales** schema. You then might also need to track orders from suppliers for product components in an **Order** table in the **Production** schema.



Database systems such as SQL Server use a hierarchical naming system. This multi-level naming helps to disambiguate tables with the same name in different schemas. The *fully qualified* name of an object includes the name of a database server instance in which the database is stored, the name of the database, the schema name, and the table name. For example: **Server1.StoreDB.Sales.Order**.

When working with tables within the context of a single database, it's common to refer to tables (and other objects) by including the schema name. For example, **Sales.Order**.

SQL Statements

In any SQL dialect, the SQL statements are grouped together into several different types of statements. These different types are:

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

Sometimes you may also see **TCL** listed as a type of statement, to refer to **Transaction Control Language**. In addition, some lists may redefine DML as **Data Modification Language**, which wouldn't include SELECT statements, but then they add **DQL** as **Data Query Language** for SELECT statements.

In this module, we'll focus on DML statements. These statements are commonly used by data analysts to retrieve data for reports and analysis. DML statements are also used by application developers to perform "CRUD" operations to create, read, update, or delete application data.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Using the SELECT Statement

The SELECT Statement

As previously mentioned, Transact-SQL or T-SQL, is a dialect of the ANSI standard SQL language used by Microsoft SQL products and services. It is similar to standard SQL. Most of our focus will be on the SELECT statement, which has by far the most options and variations of any DML statement.

Let's start by taking a high-level look at how a SELECT statement is processed. The order in which a SELECT statement is written is not the order in which it is evaluated and processed by the SQL Server database engine.

Consider the following query:

```
SELECT OrderDate, COUNT(OrderID) AS Orders
FROM Sales.SalesOrder
WHERE Status = 'Shipped'
GROUP BY OrderDate
HAVING COUNT(OrderID) > 1
ORDER BY OrderDate DESC;
```

The query consists of a SELECT statement, which is composed of multiple *clauses*, each of which defines a specific operation that must be applied to the data being retrieved. Before we examine the run-time order of operations, let's briefly take a look at what this query does, although the details of the various clauses will not be covered in this module.

The SELECT clause returns the **OrderDate** column, and the count of **OrderID** values, to which is assigns the name (or *alias*) **Orders**:

```
SELECT OrderDate, COUNT(OrderID) AS Orders
```

The FROM clause identifies which table is the source of the rows for the query; in this case it's the **Sales.SalesOrder** table:

```
FROM Sales.SalesOrder
```

The WHERE clause filters rows out of the results, keeping only those rows that satisfy the specified condition; in this case, orders that have a status of "shipped":

```
WHERE Status = 'Shipped'
```

The GROUP BY clause takes the rows that met the filter condition and groups them by**OrderDate**, so that all the rows with the same **OrderDate** are considered as a single group and one row will be returned for each group:

```
GROUP BY OrderDate
```

After the groups are formed, the HAVING clause filters the groups based on its own predicate. Only dates with more than one order will be included in the results:

```
HAVING COUNT(OrderID) > 1
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

For the purposes of previewing this query, the final clause is the ORDER BY, which sorts the output into descending order of **OrderDate**:

```
ORDER BY OrderDate DESC;
```

Now that you've seen what each clause does, let's look at the order in which SQL Server actually evaluates them:

1. The FROM clause is evaluated first, to provide the source rows for the rest of the statement. A virtual table is created and passed to the next step.
2. The WHERE clause is next to be evaluated, filtering those rows from the source table that match a predicate. The filtered virtual table is passed to the next step.
3. GROUP BY is next, organizing the rows in the virtual table according to unique values found in the GROUP BY list. A new virtual table is created, containing the list of groups, and is passed to the next step. From this point in the flow of operations, only columns in the GROUP BY list or aggregate functions may be referenced by other elements.
4. The HAVING clause is evaluated next, filtering out entire groups based on its predicate. The virtual table created in step 3 is filtered and passed to the next step.
5. The SELECT clause finally executes, determining which columns will appear in the query results. Because the SELECT clause is evaluated after the other steps, any column aliases (in our example, **Orders**) created there cannot be used in the GROUP BY or HAVING clause.
6. The ORDER BY clause is the last to execute, sorting the rows as determined by its column list.

To apply this understanding to our example query, here is the logical order at run time of the SELECT statement above:

```
FROM Sales.SalesOrder
WHERE Status = 'Shipped'
GROUP BY OrderDate
HAVING COUNT(OrderID) > 1
SELECT OrderDate, COUNT(OrderID) AS Orders
ORDER BY OrderDate DESC;
```

Not all the possible clauses are required in every SELECT statement that you write. The only required clause is the SELECT clause, which can be used on its own in some cases. Usually a FROM clause is also included to identify the table being queried. In addition, Transact-SQL has other clauses that can be added.

As you have seen, you do not write T-SQL queries in the same order in which they are logically evaluated. The run-time order of evaluation determines what data is available to which clauses, as a clause only has access to information already made available from an already processed clause. For this reason, it's important to understand the true logical processing order when writing queries.

Basic SELECT Query Examples

The SELECT clause is often referred to as the SELECT *list*, because it lists the values to be returned in the query's results.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Selecting all columns

The simplest form of a SELECT clause is the use of the asterisk character (*) to return all columns. When used in T-SQL queries, it is called a *star*. While SELECT * is suitable for a quick test, you should avoid using it in production work for the following reasons:

- Changes to the table that add or rearrange columns will be reflected in the query results, which may result in unexpected output for applications or reports that use the query.
- Returning data that is not needed can slow down your queries and cause performance issues if the source table contains a large number of rows.

For example, the following example retrieves all columns from the (hypothetical) **Production.Product** table.

```
SELECT * FROM Production.Product;
```

The result from this query is a rowset that contains all columns for all rows of the table, which might look something like this:

Produc-tID	Name	Product-Num	Color	Stand-ardCost	ListPrice	Size	Weight	Product-CatID
680	HL Road Frame - Black, 58	FR-R92B-58	Black	1059.31	1431.5	58	1016.04	18
706	HL Road Frame - Red, 58	FR-R92R-58	Red	1059.31	1431.5	58	1016.04	18
707		HL-U509-R	Red	13.0863	34.99			35
708		HL-U509	Black	13.0863	34.99			35
...	...	...	...	...	...	...	...	...

Selecting specific columns

An explicit column list allows you to have control over exactly which columns are returned and in which order. Each column in the result will have the name of the column as the header.

For example, consider the following query; which again uses the hypothetical **Production.Product** table.

```
SELECT ProductID, Name, ListPrice, StandardCost
FROM Production.Product;
```

This time, the results include only the specified columns:

ProductID	Name	ListPrice	StandardCost
680	HL Road Frame - Black, 58	1431.5	1059.31
706	HL Road Frame - Red, 58	1431.5	1059.31
707	Sport-100 Helmet, Red	34.99	13.0863

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

ProductID	Name	ListPrice	StandardCost
708	Sport-100 Helmet, Black	34.99	13.0863
...	...	...	...

Selecting expressions

In addition to retrieving columns stored in the specified table, a SELECT clause can perform calculations and manipulations, which use operators to combine columns and values or multiple columns. The result of the calculation or manipulation must be a single-valued (scalar) result that will appear in the result as a separate column.

For example, the following query includes two expressions:

```
SELECT ProductID,
       Name + '(' + ProductNumber + ')',
   ListPrice - StandardCost
FROM Production.Product;
```

The results from this query might look something like this:

ProductID		
680	HL Road Frame - Black, 58(FR-R92B-58)	372.19
706	HL Road Frame - Red, 58(FR-R92R-58)	372.19
707	Sport-100 Helmet, Red(HL-U509-R)	21.9037
708	Sport-100 Helmet, Black(HL-U509)	21.9037
...	...	...

There are a couple of interesting things to note about these results:

- The columns returned by the two expressions have no column names. Depending on the tool you are using to submit your query, a missing column name might be indicated by a blank column header, a literal *"no column name"* indicator, or a default name like **column1**. We'll see how to specify an *alias* for the column name in the query later in this section.
- The first expression uses the **+** operator to concatenate string (character-based) values, while the second expression uses the **-** operator to subtract one numeric value from another. When used with numeric values, the **+** operator performs addition. Clearly then, it is important to understand the *data types* of the columns you include in expressions. We'll discuss data types in the next section.

Specifying column aliases

You can specify an *alias* for each column returned by the SELECT query, either as an alternative to the source column name or to assign a name to the output of an expression.

For example, here's the same query as before, but with aliases specified for each of the columns:

```
SELECT ProductID AS ID,
       Name + '(' + ProductNumber + ')' AS ProductName,
   ListPrice - StandardCost AS Markup
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

```
FROM Production.Product;
```

The results from this query include the specified column names:

ID	ProductName	Markup
680	HL Road Frame - Black, 58(FR-R92B-58)	372.19
706	HL Road Frame - Red, 58(FR-R92R-58)	372.19
707	Sport-100 Helmet, Red(HL-U509-R)	21.9037
708	Sport-100 Helmet, Black(HL-U509)	21.9037
...	...	...

Note: The AS keyword is optional when specifying an alias, but it's good practice to include it for clarification.

Formatting queries

You may note from the examples in this section that you can be flexible about how you format your query code. For example, you can write each clause (or the entire query) on a single line, or break it over multiple lines. In most database systems, the code is case-insensitive, and some elements of the T-SQL language are optional (including the AS keyword as mentioned previously, and even the semi-colon at the end of a statement).

Consider the following guidelines to make your T-SQL code easily readable (and therefore easier to understand and debug!):

- Capitalize T-SQL keywords, like SELECT, FROM, AS, and so on. Capitalizing keywords is a commonly used convention that makes it easier to find each clause of a complex statement.
- Start a new line for each major clause of a statement.
- If the SELECT list contains numerous columns, expressions, or aliases, consider listing each column on its own line.
- If the SELECT list contains more than a few columns, expressions, or aliases, consider listing each column on its own line.
- Indent lines containing subclauses or columns to make it clear which code belongs to each major clause.

Data Types

Columns and variables used in Transact-SQL each have a *data type*. The behavior of values in expressions depends on the data type of the column or variable being referenced. For example, as you saw previously, you can use the **+** operator to concatenate two string values, or to add two numeric values.

The following table shows common data types supported in a SQL Server database.

Exact Numeric	Approximate Numeric	Character	Date/Time	Binary	Other
tinyint	float	char	date	binary	cursor

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Exact Numeric	Approximate Numeric	Character	Date/Time	Binary	Other
smallint	real	varchar	time	varbinary	hierarchyid
int		text	datetime	image	sql_variant
bigint		nchar	datetime2		table
bit		nvarchar	smalldatetime		timestamp
decimal/ numeric		ntext	datetimeoffset		uniqueidentifier
numeric					xml
money					geography
smallmoney					geometry

Converting data types

Values of compatible data types can be implicitly converted as required. For example, suppose you can use the **+** operator to add an *integer* number to a *decimal* number, or to concatenate a fixed-length *char* value and a variable length *varchar* value. However, in some cases you may need to explicitly convert values from one data type to another - for example, trying to use **+** to concatenate a *varchar* value and a *decimal* value will result in an error unless you first convert the numeric value to a compatible string data type.

T-SQL includes functions to help you convert between data types:

CAST and TRY_CAST

The CAST function converts a value to a specified data type if the value is compatible with the target data type. If it is not compatible, an error is returned.

For example, the following query uses CAST to convert the *integer* values in the **ProductID** column to *varchar* values (with a maximum of 4 characters) in order to concatenate them with another character-based value:

```
SELECT CAST(ProductID AS varchar(4)) + ': ' + Name AS ProductName
FROM Production.Product;
```

Possible result from this query might look something like this:

ProductName
680: HL Road Frame - Black, 58
706: HL Road Frame - Red, 58
707: Sport-100 Helmet, Red
708: Sport-100 Helmet, Black
...

However, let's suppose the **Size** column in the **Production.Product** table is a *nvarchar* column that contains some numeric sizes (like 58) and some text-based sizes (like “S”, "M", or “L”). The following query tries to convert values from this column to an integer* data type:

```
SELECT CAST(Size AS integer) As NumericSize
FROM Production.Product;
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

This query results in the following error message:

Error: Conversion failed when converting the nvarchar value 'M' to data type int.

Given that at least *some* of the values in the column are numeric, you might want to convert those values and ignore the others. You can use the TRY_CAST function to convert data types.

```
SELECT TRY_CAST(Size AS integer) As NumericSize
FROM Production.Product;
```

The results this time look might like this:

NumericSize
58
58
NULL
NULL
...

The values that can be converted to a numeric data type are returned as *decimal* values, and the incompatible values are returned as *NULL*, which is used to indicate that a value is *unknown*.

Note: We'll explore considerations for handling *NULL* values in the next section.

CONVERT and TRY_CONVERT

CAST is the ANSI standard SQL function for converting between data types, and is used in many database systems. In Transact-SQL, you can also use the CONVERT function, as shown here:

```
SELECT CONVERT(varchar(4), ProductID) + ': ' + Name AS ProductName
FROM Production.Product;
```

Once again, this query returns the value converted to the specified data type, like this:

ProductName
680: HL Road Frame - Black, 58
706: HL Road Frame - Red, 58
707: Sport-100 Helmet, Red
708: Sport-100 Helmet, Black
...

Like CAST, CONVERT has a TRY_CONVERT variant that returns *NULL* for incompatible values.

Another benefit of using CONVERT over CAST, is that CONVERT also includes a parameter that enables you specify a format style when converting numeric and date values to strings. For example, consider the following query:

```
SELECT SellStartDate,
       CONVERT(varchar(20), SellStartDate) AS StartDate,
   CONVERT(varchar(10), SellStartDate, 101) AS FormattedStartDate
FROM SalesLT.Product;
```

The results from this query might look something like this:

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

SellStartDate	StartDate	FormattedStartDate
2002-06-01T00:00:00.0000000	Jun 1 2002 12:00AM	6/1/2002
2002-06-01T00:00:00.0000000	Jun 1 2002 12:00AM	6/1/2002
2005-07-01T00:00:00.0000000	Jul 1 2005 12:00AM	7/1/2005
2005-07-01T00:00:00.0000000	Jul 1 2005 12:00AM	7/1/2005
...	...	...

Note To find out more about *style* formatting codes you can use with CONVERT, see the **Transact-SQL reference documentation**[1].

PARSE and TRY_PARSE

The PARSE function is designed to convert formatted strings that represent numeric or date/time values. For example, consider the following query (which uses literal values rather than values from columns in a table):

```
SELECT PARSE('01/01/2021' AS date) AS DateValue,
   PARSE('$199.99' AS money) AS MoneyValue;
```

The results of this query look like this:

DateValue	MoneyValue
2021-01-01T00:00:00.0000000	199.99

Similarly to CAST and CONVERT, PARSE has a TRY_PARSE variant that returns incompatible values as *NULL*.

STR

The STR function converts a numeric value to a *varchar*.

For example:

```
SELECT ProductID,  '$' + STR(ListPrice) AS Price
FROM Production.Product;
```

The results would look something like this:

ProductID	Price
680	$1432.00
706	$1432.00
707	$35.00
...	...

NULL Values

A NULL value means *no value* or *unknown*. It does not mean zero or blank, or even an empty string. Those values are not unknown. A NULL value can be used for values that haven't been supplied yet, for example, when a customer has not yet supplied an email address. As you've seen previously, a NULL

1 https://docs.microsoft.com/sql/t-sql/functions/cast-and-convert-transact-sql

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

value can also be returned by some conversion functions if a value is not compatible with the target data type.

You'll often need to take special steps to deal with NULL. NULL is really a non-value. It is unknown. It isn't equal to anything, and it's not unequal to anything. NULL isn't greater or less than anything. We can't say anything about what it is, but sometimes we need to work with NULL values. Thankfully, T-SQL provides functions for conversion or replacement of NULL values.

ISNULL

The ISNULL function takes two arguments. The first is an expression we are testing. If the value of that first argument is NULL, the function returns the second argument. If the first expression is not null, it is returned unchanged.

For example, suppose the **Sales.Customer** table in a database includes a **MiddleName** column that allows NULL values. When querying this table, rather than returning NULL in the result, you may choose to return a specific value, such as "None".

```
SELECT FirstName,
       ISNULL(MiddleName, 'None') AS MiddleIfAny,
       LastName
FROM Sales.Customer;
```

The results from this query might look something like this:

FirstName	MiddleIfAny	LastName
Orlando	N.	Gee
Keith	None	Howard
Donna	F.	Gonzales
...	...	...

Note: The value substituted for NULL must be the same datatype as the expression being evaluated. In the above example, **MiddleName** is a *varchar*, so the replacement value could not be numeric. In addition, you'll need to choose a value that will not appear in the data as a regular value. It can sometimes be difficult to find a value that will never appear in your data.

The previous example handled a NULL value in the source table, but you can use ISNULL with any expression that might return a NULL, including nesting a TRY_CONVERT function within an ISNULL function.

COALESCE

The ISNULL function is not ANSI standard, so you may wish to use the COALESCE function instead. COALESCE is a little more flexible is that it can take a variable number of arguments, each of which is an expression. It will return the first expression in the list that is not NULL.

If there are only two arguments, COALESCE behaves like ISNULL. However, with more than two arguments, COALESCE can be used as an alternative to a multipart CASE expression using ISNULL.

If all arguments are NULL, COALESCE returns NULL. All the expressions must return the same or compatible data types.

The syntax is as follows:

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

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

The following example uses a fictitious table called **HR.Wages**, which includes three columns that contain information about the weekly earnings of the employees: the hourly rate, the weekly salary, and a commission per unit sold. However, an employee receives only one type of pay. For each employee, one of those three columns will have a value, the other two will be NULL. To determine the total amount paid to each employee, you can use COALESCE to return only the non-null value found in those three columns.

```
SELECT EmployeeID,
       COALESCE(HourlyRate * 40,
                WeeklySalary,
                Commission * SalesQty) AS WeeklyEarnings
FROM HR.Wages;
```

The results might look something like this:

EmployeeID	WeeklyEarnings
1	899.76
2	1001.00
3	1298.77
...	...

NULLIF

The NULLIF function allows you to return NULL under certain conditions. This function has useful applications in areas such as data cleansing, when you wish to replace blank or placeholder characters with NULL.

NULLIF takes two arguments and returns NULL if they're equivalent. If they aren't equal, NULLIF returns the first argument.

In this example, NULLIF replaces a discount of 0 with a NULL. It returns the discount value if it is not 0:

```
SELECT SalesOrderID,
       ProductID,
       UnitPrice,
       NULLIF(UnitPriceDiscount, 0) AS Discount
FROM Sales.SalesOrderDetail;
```

The results might look something like this:

SalesOrderID	ProductID	UnitPrice	Discount
71774	836	356.898	NULL
71780	988	112.998	0.4
71781	748	818.7	NULL
71781	985	112.998	0.4
...	...	...	...

Lab - Get Started with Transact-SQL

Estimated time: 30 minutes

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Depending on the lab hosting solution used for your class delivery, the lab instructions may display directly in the virtual lab environment. In addition, they are available **online in GitHub**[2]. Follow guidance from your instructor to access the lab environment.

Module-Review

Module Review

Use the questions below to check your knowledge of the topics covered in this module.

Multiple choice

You must return the Name and Price columns from a table named Product in the Production schema. In the resulting rowset, you want the Name column to be named ProductName. Which of the following Transact-SQL statements should you use?

- ☐ SELECT * FROM Product AS Production.Product;
- ☐ SELECT Name AS ProductName, Price FROM Production.Product;
- ☐ SELECT ProductName, Price FROM Production.Product;

Multiple choice

You must retrieve data from a column that is defined as char(1). If the value in the column is a digit between 0 and 9, the query should return it as an integer value. Otherwise, the query should return NULL. Which function should you use?

- ☐ CAST
- ☐ NULLIF
- ☐ TRY_CONVERT

Multiple choice

You must return the Cellphone column from the Sales.Customer table. Cellphone is a varchar column that permits NULL values. For rows where the Cellphone value is NULL, your query should return the text 'None'. What query should you use?

- ☐ SELECT ISNULL(Cellphone, 'None') AS Cellphone FROM Sales.Customer;
- ☐ SELECT NULLIF(Cellphone, 'None') AS Cellphone FROM Sales.Customer;
- ☐ SELECT CONVERT(varchar, Cellphone) AS None FROM Sales.Customer;

2 https://microsoftlearning.github.io/dp-080-Transact-SQL/

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Answers

Multiple choice

You must return the Name and Price columns from a table named Product in the Production schema. In the resulting rowset, you want the Name column to be named ProductName. Which of the following Transact-SQL statements should you use?

- ☐ SELECT * FROM Product AS Production.Product;
- ■ SELECT Name AS ProductName, Price FROM Production.Product;
- ☐ SELECT ProductName, Price FROM Production.Product;

Explanation
Select Name and Price from Production.Product table, using the AS keyword to specify the alias Product-Name for the Name column.

Multiple choice

You must retrieve data from a column that is defined as char(1). If the value in the column is a digit between 0 and 9, the query should return it as an integer value. Otherwise, the query should return NULL. Which function should you use?

- ☐ CAST
- ☐ NULLIF
- ■ TRY_CONVERT

Explanation
Use TRY_CONVERT to convert the value to an integer. If the conversion fails, NULL will be returned.

Multiple choice

You must return the Cellphone column from the Sales.Customer table. Cellphone is a varchar column that permits NULL values. For rows where the Cellphone value is NULL, your query should return the text 'None'. What query should you use?

- ■ SELECT ISNULL(Cellphone, 'None') AS Cellphone FROM Sales.Customer;
- ☐ SELECT NULLIF(Cellphone, 'None') AS Cellphone FROM Sales.Customer;
- ☐ SELECT CONVERT(varchar, Cellphone) AS None FROM Sales.Customer;

Explanation
Use ISNULL to return the specified value when the target column is NULL.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Module 2 Sorting and Filtering Query Results

Sorting Query Results

Sorting Results

In the logical order of query processing, ORDER BY is the last phase of a SELECT statement to be executed. ORDER BY enables you to control the sorting of rows as they are returned from SQL Server to the client application. SQL Server doesn't guarantee the physical order of rows in a table, and the only way to control the order the rows will be returned to the client is with an ORDER BY clause. This behavior is consistent with relational theory.

Using the ORDER BY Clause

To tell SQL Server to return the results of your query in a particular order, you add an ORDER BY clause in this form:

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

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

- **Columns by name**. You can specify the names of the column(s) by which the results should be sorted. The results are returned in order of the first column, and then subsorted by each additional column in order.
- **Column aliases**. Because the ORDER BY is processed after the SELECT clause, it has access to aliases defined in the SELECT list.
- **Columns by ordinal position in the SELECT list**. Using the position isn't recommended in your applications, because of diminished readability and the extra care required to keep the ORDER BY list up to date. However, for complex expressions in the SELECT list, using the position number can be useful during troubleshooting.

- **Columns not included in the SELECT list, but available from tables listed in the FROM clause**. If the query uses a DISTINCT option, any columns in the ORDER BY list must be included in the SELECT list.

Sort direction

In addition to specifying which columns should be used to determine the sort order, you may also control the direction of the sort. You can use ASC for ascending (A-Z, 0-9) or DESC for descending (Z-A, 9-0). Ascending sorts are the default. Each column can have its own direction specified, as in the following example:

```
SELECT ProductCategoryID AS Category, ProductName
FROM Production.Product
ORDER BY Category ASC, Price DESC;
```

Limiting Sorted Results

The TOP clause is a Microsoft-proprietary extension of the SELECT clause. TOP will let you specify how many rows to return, either as a positive integer or as a percentage of all qualifying rows. The number of rows can be specified as a constant or as an expression. TOP is most frequently used with an ORDER BY, but can be used with unordered data.

Using the TOP clause

The simplified syntax of the TOP clause, used with ORDER BY, is as follows:

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

For example, to retrieve only the 10 most expensive products from the **Production.Product** table, use the following query:

```
SELECT TOP 10 Name, ListPrice
FROM Production.Product
ORDER BY ListPrice DESC;
```

The results might look something like this:

Name	ListPrice
Road-150 Red, 62	3578.27
Road-150 Red, 44	3578.27
Road-150 Red, 48	3578.27
Road-150 Red, 52	3578.27
Road-150 Red, 56	3578.27
Mountain-100 Silver, 38	3399.99
Mountain-100 Silver, 42	3399.99
Mountain-100 Silver, 44	3399.99

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Name	ListPrice
Mountain-100 Silver, 48	3399.99
Mountain-100 Black, 38	3374.99

The TOP operator depends on an ORDER BY clause to provide meaningful precedence to the rows selected. TOP can be used without ORDER BY, but in that case, there is no way to predict which rows will be returned. In this example, any 10 orders might be returned if there wasn't an ORDER BY clause.

Using WITH TIES

In addition to specifying a fixed number of rows to be returned, the TOP keyword also accepts the WITH TIES option, which will retrieve any rows with values that might be found in the selected top N rows.

In the previous example, the query returned the first 10 products in descending order of price. However, by adding the WITH TIES option to the TOP clause, you will see that more rows qualify for inclusion in the top 10 most expensive products:

```
SELECT TOP 10 WITH TIES Name, ListPrice
FROM Production.Product
ORDER BY ListPrice DESC;
```

This modified query returns the following results:

Name	ListPrice
Road-150 Red, 62	3578.27
Road-150 Red, 44	3578.27
Road-150 Red, 48	3578.27
Road-150 Red, 52	3578.27
Road-150 Red, 56	3578.27
Mountain-100 Silver, 38	3399.99
Mountain-100 Silver, 42	3399.99
Mountain-100 Silver, 44	3399.99
Mountain-100 Silver, 48	3399.99
Mountain-100 Black, 38	3374.99
Mountain-100 Black, 42	3374.99
Mountain-100 Black, 44	3374.99
Mountain-100 Black, 48	3374.99

The decision to include WITH TIES will depend on your knowledge of the source data, its potential for unique values, and the requirements of the query you are writing.

Using PERCENT

To return a percentage of the eligible rows, use the PERCENT option with TOP instead of a fixed number.

```
SELECT TOP 10 PERCENT Name, ListPrice
FROM SalesLT.Product
ORDER BY ListPrice DESC;
```

The PERCENT may also be used with the WITH TIES option.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

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

The TOP option is used by many SQL Server professionals as a method for retrieving only a certain range of rows. However, consider the following facts when using TOP:

- TOP is proprietary to T-SQL.
- TOP does not support skipping a range of rows.
- Because TOP depends on an ORDER BY clause, you cannot use one sort order to establish the rows filtered by TOP and another to determine the output order.

Paging Through Results

An extension to the ORDER BY clause called OFFSET-FETCH enables you to return only a range of the rows selected by your query. It adds the ability to supply a starting point (an offset) and a value to specify how many rows you would like to return (a fetch value). This extension provides a convenient technique for paging through results.

If you want to return rows a "page" at a time (using whatever number you choose for a page), you'll need to consider that each query with an OFFSET-FETCH clause runs independently of any other queries. There's no server-side state maintained, and you'll need to track your position through a result set via client-side code.

OFFSET-FETCH Syntax

The syntax for the OFFSET-FETCH clause, which is technically part of the ORDER BY clause, is as follows:

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

Using OFFSET-FETCH

To use OFFSET-FETCH, you'll supply a starting OFFSET value, which may be zero, and an optional number of rows to return, as in the following example:

This example will return the first 10 rows, and then return the next 10 rows of product data based on the **ListPrice**:

```
SELECT ProductID, ProductName, ListPrice
FROM Production.Product
ORDER BY ListPrice DESC
OFFSET 0 ROWS--Skip zero rows
FETCH NEXT 10 ROWS ONLY; -- Get the next 10
```

To retrieve the *next* page of product data, use the OFFSET clause to specify the number of rows to skip:

```
SELECT ProductID, ProductName, ListPrice
FROM Production.Product
ORDER BY ListPrice DESC
OFFSET 10 ROWS--Skip 10 rows
FETCH NEXT 10 ROWS ONLY; -- Get the next 10
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

In the syntax definition you can see the OFFSET clause is required, but the FETCH clause is not. If the FETCH clause is omitted, all rows following OFFSET will be returned. You'll also find that the keywords ROW and ROWS are interchangeable, as are FIRST and NEXT, which enables a more natural syntax.

To ensure the accuracy of the results, especially as you move from page to page of data, it's important to construct an ORDER BY clause that will provide unique ordering and yield a deterministic result. Because of the way SQL Server's query optimizer works, it's technically possible for a row to appear on more than one page, unless the range of rows is deterministic.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Filtering Query Results

Removing Duplicates

Although the rows in a table should always be unique, when you select only a subset of the columns, the result rows may not be unique even if the original rows are. For example, you may have a table of suppliers with a requirement the city and state (or province) be unique so that there will never be more than one supplier in any city. However, if you just want to see the cities and countries/regions where suppliers are located, the returned results may not be unique. Suppose you write the following query:

```
SELECT City, CountryRegion
FROM Production.Supplier
ORDER BY CountryRegion, City;
```

This query may return results similar to the following:

City	CountryRegion
Aurora	Canada
Barrie	Canada
Brampton	Canada
Brossard	Canada
Brossard	Canada
Burnaby	Canada
Burnaby	Canada
Burnaby	Canada
Calgary	Canada
Calgary	Canada
...	...

By default, the SELECT clause includes an implicit ALL keyword that results in this behavior:

```
SELECT ALL City, CountryRegion
FROM Production.Supplier
ORDER BY CountryRegion, City;
```

T-SQL also supports an alternative the DISTINCT keyword, which removes any duplicate result rows:

```
SELECT DISTINCT City, CountryRegion
FROM Production.Supplier
ORDER BY CountryRegion, City;
```

When using DISTINCT, the example returns only one of each unique combination of values in the SELECT list:

City	CountryRegion
Aurora	Canada
Barrie	Canada
Brampton	Canada
Brossard	Canada

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

City	CountryRegion
Burnaby	Canada
Calgary	Canada
...	...

Filtering and Using Predictates

The simplest SELECT statements with just SELECT and FROM clauses will evaluate every row in a table. If you only want a subset of rows to be processed, you can include a WHERE clause, which defines conditions that determine which rows will be processed and returned.

The structure of the WHERE clause

The WHERE clause is made up of one or more search conditions, each of which must evaluate to TRUE, FALSE or 'unknown' for each row of the table. Only rows for which the WHERE clause evaluates to TRUE will be returned. The individual conditions are frequently referred to as 'predicates' and they act as filters on the data rows. Each predicate includes a condition that is being tested, usually using the basic comparison operators:

- = (equals)
- <> (not equals)
- > (greater than)
- >= (greater than or equal to)
- < (less than)
- <= (less than or equal to)

For example, the following query returns all products with a **ProductCategoryID** value of 2:

```
SELECT ProductCategoryID AS Category, ProductName
FROM Production.Product
WHERE ProductCategoryID = 2;
```

Similarly, the following query returns all products with a **ListPrice** less than 10.00:

```
SELECT ProductCategoryID AS Category, ProductName
FROM Production.Product
WHERE ListPrice < 10.00;
```

Multiple conditions

Multiple predicates can be combined with the AND and OR operators, and with parentheses. However SQL Server will only process two conditions at a time. The result of two conditions connected with AND will only be TRUE (and the row will be returned) if both conditions are TRUE. The result of two conditions connected with OR will be TRUE if either one, or both, of the conditions are TRUE.

For example, the following query returns product in category 2 that cost less than 10.00:

```
SELECT ProductCategoryID AS Category, ProductName
FROM Production.Product
WHERE ProductCategoryID = 2
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

```
AND ListPrice < 10.00;
```

AND operators are processed before OR operators unless parentheses are used. Best practice, however, dictates that if you have more than two predicates, you should use parenthesis to avoid confusion, as shown in the following query, which returns products in category 2 or 3 that cost less than 10.00:

```
SELECT ProductCategoryID AS Category, ProductName
FROM Production.Product
WHERE (ProductCategoryID = 2 OR ProductCategoryID = 3)
AND (ListPrice < 10.00);
```

Comparison operators

Transact-SQL provides some additional comparison operators that can help simplify the WHERE clause.

IN

The IN operator is a shortcut for multiple equality conditions for the same column connected with OR. There is nothing incorrect about writing a query with multiple OR conditions, as in the following example:

```
SELECT ProductCategoryID AS Category, ProductName
FROM Production.Product
WHERE ProductCategoryID = 2
    OR ProductCategoryID = 3
    OR ProductCategoryID = 4;
```

However, using IN is a bit clearer, and saves some typing. The performance of the query will not be affected.

```
SELECT ProductCategoryID AS Category, ProductName
FROM Production.Product
WHERE ProductCategoryID IN (2, 3, 4);
```

BETWEEN

Another shortcut can sometimes be used when there are two inequality comparisons on the same column combined with AND, that result in an upper and lower bound for the value. The following two queries are equivalent:

```
SELECT ProductCategoryID AS Category, ProductName
FROM Production.Product
WHERE ListPrice >= 1.00
    AND ListPrice <= 10.00;
```

```
SELECT ProductCategoryID AS Category, ProductName
FROM Production.Product
WHERE ListPrice BETWEEN 1.00 AND 10.00;
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

The BETWEEN operator uses inclusive boundary values. Products with a price of either 1.00 or 10.00 would be included in the results.

LIKE

The final comparison operator can only be used for character data and allows us to use wildcard characters and regular expression patterns. Wildcards allow us to specify partial strings. For example, you could use the following query to return all products with names that contain the word "mountain":

```
SELECT Name, ListPrice
FROM SalesLT.Product
WHERE Name LIKE '%mountain%';
```

The **%** wildcard represents any string of 0 or more characters, so the results include products with the word "mountain" anywhere in their name, like this:

Name	ListPrice
Mountain Bike Socks, M	9.50
Mountain Bike Socks, L	9.50
HL Mountain Frame - Silver, 42	1364.0
HL Mountain Frame - Black, 42	1349.60
HL Mountain Frame - Silver, 38	1364.50
Mountain-100 Silver, 38	3399.99
...	...

You can use the _ wildcard to represent a single character, like this:

```
SELECT Name, ListPrice
FROM SalesLT.Product
WHERE Name LIKE 'Mountain Bike Socks, _';
```

The results of this query include products with a name that begins with "Mountain Bike Socks, ", and is followed by a single character:

Name	ListPrice
Mountain Bike Socks, M	9.50
Mountain Bike Socks, L	9.50

You can define complex patterns for strings that you want to find. For example, the following query searched for products with a name that starts with "Mountain-", followed by three characters between 0 and 9, followed by a space and then any string, and ending with a comma, a space, and two characters between 0 and 9.

```
SELECT Name, ListPrice
FROM SalesLT.Product
WHERE Name LIKE 'Mountain-[0-9][0-9][0-9] %, [0-9][0-9]';
```

The results from this query might look something like this:

Name	ListPrice
Mountain-100 Silver, 38	3399.99

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Name	ListPrice
Mountain-100 Silver, 42	3399.99
Mountain-100 Black, 38	3399.99
Mountain-100 Black, 42	3399.99
Mountain-200 Silver, 38	2319.99
Mountain-200 Silver, 42	2319.99
Mountain-200 Black, 38	2319.99
Mountain-200 Black, 42	2319.99
...	...

Lab - Sort and Filter Query Results

Estimated time: 30 minutes

Depending on the lab hosting solution used for your class delivery, the lab instructions may display directly in the virtual lab environment. In addition, they are available **online in GitHub**[1]. Follow guidance from your instructor to access the lab environment.

Module-Review

Module Review

Use the questions below to check your knowledge of the topics covered in this module.

Multiple choice

You write a Transact-SQL query to list the available sizes for products. Each individual size should be listed only once. Which query should you use?

- ☐ SELECT Size FROM Production.Product;
- ☐ SELECT DISTINCT Size FROM Production.Product;
- ☐ SELECT ALL Size FROM Production.Product;

Multiple choice

You must return the InvoiceNo and TotalDue columns from the Sales.Invoice table in decreasing order of TotalDue value. Which query should you use?

- ☐ SELECT * FROM Sales.Invoice ORDER BY TotalDue, InvoiceNo;
- ☐ SELECT InvoiceNo, TotalDue FROM Sales.Invoice ORDER BY TotalDue DESC;
- ☐ SELECT TotalDue AS DESC, InvoiceNo FROM Sales.Invoice;

1 https://microsoftlearning.github.io/dp-080-Transact-SQL/

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Multiple choice

Complete this query to return only products that have a Category value of 2 or 4: SELECT Name, Price FROM Production.Product Which clause should you add?

- ☐ ORDER BY Category;
- ☐ WHERE Category BETWEEN 2 AND 4;
- ☐ WHERE Category IN (2, 4);

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Answers

Multiple choice

You write a Transact-SQL query to list the available sizes for products. Each individual size should be listed only once. Which query should you use?

- ☐ SELECT Size FROM Production.Product;
- ■ SELECT DISTINCT Size FROM Production.Product;
- ☐ SELECT ALL Size FROM Production.Product;

Explanation
Use the DISTINCT keyword to eliminate duplicate rows in the resultset.

Multiple choice

You must return the InvoiceNo and TotalDue columns from the Sales.Invoice table in decreasing order of TotalDue value. Which query should you use?

- ☐ SELECT * FROM Sales.Invoice ORDER BY TotalDue, InvoiceNo;
- ■ SELECT InvoiceNo, TotalDue FROM Sales.Invoice ORDER BY TotalDue DESC;
- ☐ SELECT TotalDue AS DESC, InvoiceNo FROM Sales.Invoice;

Explanation
Use ORDER BY with DESC to sort the results in descending order.

Multiple choice

Complete this query to return only products that have a Category value of 2 or 4: SELECT Name, Price FROM Production.Product Which clause should you add?

- ☐ ORDER BY Category;
- ☐ WHERE Category BETWEEN 2 AND 4;
- ■ WHERE Category IN (2, 4);

Explanation
Use WHERE with an IN clause to filter based on discrete values in a list.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Module 3 Using Joins and Subqueries

Using Joins

Join Concepts

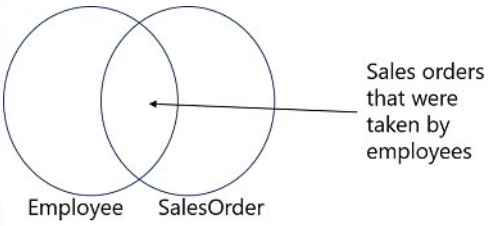
The most fundamental and common method of combining data from multiple tables is to use a JOIN operation. Some people think of JOIN as a separate clause in a SELECT statement, but others think of it as part of the FROM clause. This module will mainly consider it to be part of the FROM clause. In this module, we'll discover how the FROM clause in a T-SQL SELECT statement creates intermediate virtual tables that will be consumed by later phases of the query.

The FROM Clause and Virtual Tables

If you've learned about the logical order of operations that are performed when SQL Server processes a query, you've seen that the FROM clause of a SELECT statement is the first clause to be processed. This clause determines which table or tables will be the source of rows for the query. The FROM can reference a single table or bring together multiple tables as the source of data for your query. You can think of the FROM clause as creating and populating a virtual table. This virtual table will hold the output of the FROM clause and be used by clauses of the SELECT statement that are applied later, such as the WHERE clause. As you add extra functionality, such as join operators, to a FROM clause, it will be helpful to think of the purpose of the FROM clause elements as either to add rows to, or remove rows from, the virtual table.

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

The virtual table created by the FROM clause contains data from all of the joined tables. It can be useful to think of the results as *sets*, and conceptualize the join results as a Venn diagram.



Join Syntax

Throughout its history, the T-SQL language has expanded to reflect changes to the American National Standards Institute (ANSI) standards for the SQL language. One of the most notable places where these changes are visible is in the syntax for joins in a FROM clause. In the ANSI SQL-89 standard, joins were specified by including multiple tables in the FROM clause in a comma-separated list. Any filtering to determine which rows to include were performed in the WHERE clause, like this:

```
SELECT p.ProductID, m.Name AS Model, p.Name AS Product
FROM SalesLT.Product AS p, SalesLT.ProductModel AS m
WHERE p.ProductModelID = m.ProductModelID;
```

This syntax is still supported by SQL Server, but because of the complexity of representing the filters for complex joins, it is not recommended. Additionally, if a WHERE clause is accidentally omitted, ANSI SQL-89-style joins can easily become Cartesian products and return an excessive number of result rows, causing performance problems, and possibly incorrect results.

Note: When learning about writing multi-table queries in T-SQL, it's important to understand the concept of Cartesian products. In mathematics, a Cartesian product is the product of two sets. The product of a set of two elements and a set of six elements is a set of 12 elements, or 6 x 2. Every element in one set is combined with every element in the other set. In the example below, we have a set of names with two elements and a set of products with three elements. The Cartesian product combines every name with every product yielding six elements.

Name
Davis
Funk

Product
Alice Mutton
Crab Meat
Ipoh Coffee

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

Note: In databases, a Cartesian product is the result of combining every row in one table to every row of another table. The product of a table with 10 rows and a table with 100 rows is a result set with 1,000 rows. The underlying result of a JOIN operation is a Cartesian product but for most T-SQL queries, a Cartesian product isn't the desired result. In T-SQL, a Cartesian product occurs when two input tables are joined without considering any relationships between them. With no information about relationships, the SQL Server query processor will return all possible combinations of rows. While this result can have some

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

practical applications, such as generating test data, it's not typically useful and can have severe performance implications.

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

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

```
SELECT p.ProductID, m.Name AS Model, p.Name AS Product
FROM SalesLT.Product AS p
JOIN SalesLT.ProductModel AS m
    ON p.ProductModelID = m.ProductModelID;
```

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

Inner Joins

The most frequent type of JOIN in T-SQL queries is INNER JOIN. Inner joins are used to solve many common business problems, especially in highly normalized database environments. To retrieve data that has been stored across multiple tables, you will often need to combine it via INNER JOIN queries. An INNER JOIN begins its logical processing phase as a Cartesian product, which is then filtered to remove any rows that don't match the predicate.

Processing an INNER JOIN

Let's examine the steps by which SQL Server will logically process a JOIN query. Line numbers in the following hypothetical example are added for clarity:

```
1) SELECT emp.FirstName, ord.Amount
2) FROM HR.Employee AS emp
3) JOIN Sales.SalesOrder AS ord
4)      ON emp.EmployeeID = ord.EmployeeID;
```

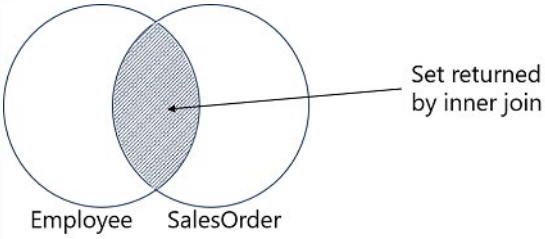
As you should be aware, the FROM clause will be processed before the SELECT clause. Let's track the processing, beginning with line 2:

- The FROM clause specifies the **HR.Employee** table as one of the input tables, giving it the alias **emp**.
- The JOIN operator in line 3 reflects the use of an INNER JOIN (the default type in T-SQL) and specifies **Sales.SalesOrder** as the other input table, which has an alias of **ord**.
- SQL Server will perform a logical Cartesian join on these tables and pass the results as a virtual table to the next step. (The physical processing of the query may not actually perform the Cartesian product operation, depending on the optimizer's decisions. But it can be helpful to imagine the Cartesian product being created.)
- Using the ON clause, SQL Server will filter the virtual table, keeping only those rows where an **EmployeeID** value from the **emp** table matches a **EmployeeID** in the **ord** table.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

- The remaining rows are left in the virtual table and handed off to the next step in the SELECT statement. In this example, the virtual table is next processed by the SELECT clause, and the three specified columns are returned to the client application.

The result of the completed query is a list of employees and their order amounts. Employees that do not have any associated orders have been filtered out by the ON clause, as have any orders that happen to have a **EmployeeID** that doesn't correspond to an entry in the **HR.Employee** table.



INNER JOIN syntax

An INNER JOIN is the default type of JOIN, and the optional INNER keyword is implicit in the JOIN clause. When mixing and matching join types, it can be useful to specify the join type explicitly, as shown in this hypothetical example:

```
SELECT emp.FirstName, ord.Amount
FROM HR.Employee AS emp
INNER JOIN Sales.SalesOrder AS ord
    ON emp.EmployeeID = ord.EmployeeID;
```

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

- Table aliases are preferred, not only for the SELECT list, but also for writing the ON clause.
- Inner joins may be performed on a single matching column, such as an **OrderID**, or on multiple matching attributes, such as the combination of **OrderID** and **ProductID**. Joins that specify multiple matching columns are called *composite* joins.
- The order in which tables are listed in the FROM clause for an INNER JOIN doesn't matter to the SQL Server optimizer. Conceptually, joins will be evaluated from left to right.
- Use the JOIN keyword once for each pair of joined tables in the FROM list. For a two-table query, specify one join. For a three-table query, you'll use JOIN twice; once between the first two tables, and once again between the output of the JOIN between the first two tables and the third table.

INNER JOIN examples

The following hypothetical example performs a join on a single matching column, relating the **ProductModelID** in the **Production.Product** table to the **ProductModelID** on the **Production.ProductModel** table:

```
SELECT p.ProductID, m.Name AS Model, p.Name AS Product
FROM Production.Product AS p
INNER JOIN Production.ProductModel AS m
    ON p.ProductModelID = m.ProductModelID
ORDER BY p.ProductID;
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

This next example shows how an inner join may be extended to include more than two tables. The **Sales.SalesOrderDetail** table is joined to the output of the JOIN between **Production.Product** and **Production.ProductModel**. Each instance of JOIN/ON does its own population and filtering of the virtual output table. The SQL Server query optimizer determines the order in which the joins and filtering will be performed.

```
SELECT od.SalesOrderID, m.Name AS Model, p.Name AS ProductName, od.OrderQty
FROM Production.Product AS p
INNER JOIN Production.ProductModel AS m
    ON p.ProductModelID = m.ProductModelID
INNER JOIN Sales.SalesOrderDetail AS od
    ON p.ProductID = od.ProductID
ORDER BY od.SalesOrderID;
```

Outer Joins

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

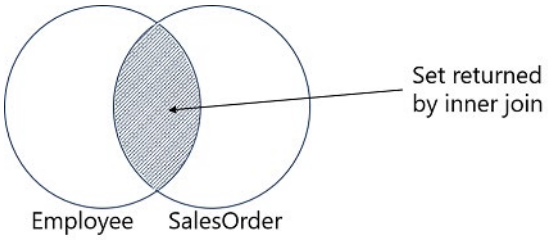
Understanding OUTER JOIN

Previously, you learned how to use an INNER JOIN to find matching rows between two tables. As you saw, the query processor builds the results of an INNER JOIN query by filtering out rows that don't meet the conditions expressed in the ON clause predicate. The result is that only rows with a matching row in the other table are returned. With an OUTER JOIN, you can choose to display all the rows that have matching rows between the tables, plus all the rows that don't have a match in the other table. Let's look at an example, then explore the process.

First, examine the following query, written with an INNER JOIN:

```
SELECT emp.FirstName, ord.Amount
FROM HR.Employee AS emp
INNER JOIN Sales.SalesOrder AS ord
    ON emp.EmployeeID = ord.EmployeeID;
```

These rows represent a match between **HR.Employee** and **Sales.SalesOrder**. Only those **EmployeeID** values that are in both tables will appear in the results.



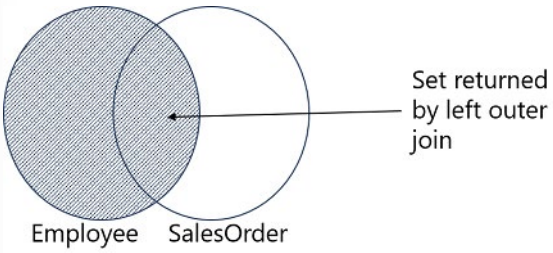
Now, let's examine the following query, written as LEFT OUTER JOIN:

```
SELECT emp.FirstName, ord.Amount
FROM HR.Employee AS emp
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

```
LEFT OUTER JOIN Sales.SalesOrder AS ord
    ON emp.EmployeeID = ord.EmployeeID;
```

This example uses a LEFT OUTER JOIN operator, which directs the query processor to preserve all rows from the table on the left (**HR.Employee**) and displays the **Amount** values for matching rows in **Sales.SalesOrder**. However, all employees are returned, whether or not they have taken a sales order. In place of the **Amount** value, the query will return NULL for employees with no matching sales orders.



OUTER JOIN Syntax

Outer joins are expressed using the keywords LEFT, RIGHT, or FULL preceding OUTER JOIN. The purpose of the keyword is to indicate which table (on which side of the keyword JOIN) should be preserved and have all its rows displayed; match, or no match.

When using LEFT, RIGHT, or FULL to define a join, you can omit the OUTER keyword as shown here:

```
SELECT emp.FirstName, ord.Amount
FROM HR.Employee AS emp
LEFT JOIN Sales.SalesOrder AS ord
    ON emp.EmployeeID = ord.EmployeeID;
```

However, like the INNER keyword, it is often helpful to write code that is explicit about the kind of join being used.

When writing queries using OUTER JOIN, consider the following guidelines:

- As you have seen, table aliases are preferred not only for the SELECT list, but also for the ON clause.
- As with an INNER JOIN, an OUTER JOIN may be performed on a single matching column or on multiple matching attributes.
- Unlike an INNER JOIN, the order in which tables are listed and joined in the FROM clause ***does*** matter with OUTER JOIN, as it will determine whether you choose LEFT or RIGHT for your join.
- Multi-table joins are more complex when an OUTER JOIN is present. The presence of NULLs in the results of an OUTER JOIN may cause issues if the intermediate results are then joined to a third table. Rows with NULLs may be filtered out by the second join's predicate.
- To display only rows where no match exists, add a test for NULL in a WHERE clause following an OUTER JOIN predicate.
- A FULL OUTER JOIN is used rarely. It returns all the matching rows between the two tables, plus all the rows from the first table with no match in the second, plus all the rows in the second without a match in the first.
- There is no way to predict the order the rows will come back without an ORDER BY clause. There's no way to know if the matched or unmatched rows will be returned first.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Cross Joins

A cross join is simply a Cartesian product of the two tables. Using ANSI SQL-89 syntax, you can create a cross join by just leaving off the filter that connects the two tables. Using the ANSI-92 syntax, it's a little harder; which is good, because in general, a cross join isn't something that you usually want. With the ANSI-92 syntax, it's highly unlikely you'll end up with a cross join accidentally.

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

This operation creates a result set with all possible combinations of input rows:

```
SELECT <select_list>
FROM table1 AS t1
CROSS JOIN table2 AS t2;
```

While this result isn't typically a desired output, there are a few practical applications for writing an explicit CROSS JOIN:

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

Cross Join Syntax

When writing queries with CROSS JOIN, consider the following guidelines:

- There is no matching of rows performed, and so no ON clause is used. (It is an error to use an ON clause with CROSS JOIN.)
- To use ANSI SQL-92 syntax, separate the input table names with the CROSS JOIN operator.

Cross Join Example

The following query is an example of using CROSS JOIN to create all combinations of employees and products:

```
SELECT emp.FirstName, prd.Name
FROM HR.Employee AS emp
CROSS JOIN Production.Product AS prd;
```

Self-Joins

So far, the joins we've used have involved different tables. There may be scenarios in which you need to retrieve and compare rows from a table with other rows from the same table. For example, in a human resources application, an **Employee** table might include information about the manager of each employee, and store the manager's ID in the employee's own row. Each manager is also listed as an employee.

EmployeeID	FirstName	ManagerID
1	Dan	NULL
2	Aisha	1
3	Rosie	1
4	Naomi	3

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

To retrieve the employee information and match it to the related manager, you can use the table twice in your query, joining it to itself for the purposes of the query.

```
SELECT emp.FirstName AS Employee,
       mgr.FirstName AS Manager
FROM HR.Employee AS emp
LEFT OUTER JOIN HR.Employee AS mgr
  ON emp.ManagerID = mgr.EmployeeID;
```

The results of this query include a row for each employee with the name of their manager. The CEO of the company has no manager. To include the CEO in the results, an outer join is used, and the manager name is returned as NULL for rows where the **ManagerID** field has no matching **EmployeeID** field.

Employee	Manager
Dan	NULL
Aisha	Dan
Rosie	Dan
Naomi	Rosie

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

To accomplish tasks like this, you should consider the following guidelines:

- Define two instances of the same table in the FROM clause, and join them as needed, using inner or outer joins.
- Use table aliases to differentiate the two instances of the same table.
- Use the ON clause to provide a filter comparing columns of one instance of the table with columns from the other instance of the table.

Lab - Query Multiple Tables with Joins

Estimated time: 30 minutes

Depending on the lab hosting solution used for your class delivery, the lab instructions may display directly in the virtual lab environment. In addition, they are available **online in GitHub**[1]. Follow guidance from your instructor to access the lab environment.

1 https://microsoftlearning.github.io/dp-080-Transact-SQL/

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Using Subqueries

Introduction to Subqueries

A subquery is a SELECT statement nested within another query. Being able to nest one query within another will enhance your ability to create effective queries in T-SQL. In general, subqueries are evaluated once, and provide their results to the outer query.

Working with Subqueries

A subquery is a SELECT statement nested, or embedded, within another query. The nested query, which is the subquery, is referred to as the inner query. The query containing the nested query is the outer query.

The purpose of a subquery is to return results to the outer query. The form of the results will determine whether the subquery is a scalar or multi-valued subquery:

- Scalar subqueries return a single value. Outer queries must process a single result.
- Multi-valued subqueries return a result much like a single-column table. Outer queries must be able to process multiple values.

In addition to the choice between scalar and multi-valued subqueries, subqueries can either be self-contained subqueries or they can be correlated with the outer query:

- Self-contained subqueries can be written as stand-alone queries, with no dependencies on the outer query. A self-contained subquery is processed once, when the outer query runs and passes its results to that outer query.
- Correlated subqueries reference one or more columns from the outer query and therefore depend on it. Correlated subqueries cannot be run separately from the outer query.

Scalar or Multi-Valued Subqueries

A scalar subquery is an inner SELECT statement within an outer query, written to return a single value. Scalar subqueries may be used anywhere in an outer T-SQL statement where a single-valued expression is permitted—such as in a SELECT clause, a WHERE clause, a HAVING clause, or even a FROM clause. They can also be used in data modification statements, such as UPDATE or DELETE.

Multi-valued subqueries, as the name suggests, can return more than one row. However they still return a single column.

Scalar subqueries

Suppose you want to retrieve the details of the last order that was placed, on the assumption that it is the one with the highest **SalesOrderID** value.

To find the highest **SalesOrderID** value, you might use the following query:

```
SELECT MAX(SalesOrderID)
FROM Sales.SalesOrderHeader
```

This query returns a single value that indicates the highest value for an **OrderID** in the **SalesOrderHeader** table.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

To get the details for this order, you might need to filter the **SalesOrderDetails** table based on whatever value is returned by the query above. You can accomplish this task by nesting the query to retrieve the maximum **SalesOrderID** within the WHERE clause of a query that retrieves the order details.

```
SELECT SalesOrderID, ProductID, OrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID =
   (SELECT MAX(SalesOrderID)
    FROM Sales.SalesOrderHeader);
```

To write a scalar subquery, consider the following guidelines:

- To denote a query as a subquery, enclose it in parentheses.
- Multiple levels of subqueries are supported in Transact-SQL. In this module, we'll only consider two-level queries (one inner query within one outer query), but up to 32 levels are supported.
- If the subquery returns no rows (an empty set), the result of the subquery is a NULL. If it is possible in your scenario for no rows to be returned, you should ensure your outer query can gracefully handle a NULL, in addition to other expected results.
- The inner query should generally return a single column. Selecting multiple columns in a subquery is almost always an error. The only exception is if the subquery is introduced with the EXISTS keyword.

A scalar subquery can be used anywhere in a query where a value is expected, including the SELECT list. For example, we could extend the query that retrieved details for the most recent order to include the average quantity of items that is ordered, so we can compare the quantity ordered in the most recent order with the average for all orders.

```
SELECT SalesOrderID, ProductID, OrderQty,
    (SELECT AVG(OrderQty)
     FROM SalesLT.SalesOrderDetail) AS AvgQty
FROM SalesLT.SalesOrderDetail
WHERE SalesOrderID =
    (SELECT MAX(SalesOrderID)
     FROM SalesLT.SalesOrderHeader);
```

Multi-Valued subqueries

A multi-valued subquery is well suited to return results using the IN operator. The following hypothetical example returns the **CustomerID**, **SalesOrderID** values for all orders placed by customers in Canada.

```
SELECT CustomerID, SalesOrderID
FROM Sales.SalesOrderHeader
WHERE CustomerID IN (
    SELECT CustomerID
    FROM Sales.Customer
    WHERE CountryRegion = 'Canada');
```

In this example, if you were to execute only the inner query, a column of **CustomerID** values would be returned, with a row for each customer in Canada.

In many cases, multi-valued subqueries can easily be written using joins. For example, here's a query that uses a join to return the same results as the previous example:

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

```
SELECT c.CustomerID, o.SalesOrderID
FROM Sales.Customer AS c
JOIN Sales.SalesOrderHeader AS o
    ON c.CustomerID = o.CustomerID
WHERE c. CountryRegion = 'Canada';
```

So how do you decide whether to write a query involving multiple tables as a JOIN or with a subquery? Sometimes, it just depends on what you're more comfortable with. Most nested queries that are easily converted to JOINs will actually BE converted to a JOIN internally. For such queries, there is then no real difference in writing the query one way vs another.

One restriction you should keep in mind is that when using a nested query, the results returned to the client can only include columns from the outer query. So if you need to return columns from both tables, you should write the query using a JOIN.

Finally, there are situations where the inner query needs to perform much more complicated operations that the simple retrievals in our examples. Rewriting complex subqueries using a JOIN can be difficult. Many SQL developers find subqueries work best for complicated processing because it allows you to break down the processing into smaller steps.

Self-Contained or Correlated Subqueries?

Previously, we looked at self-contained subqueries; in which the inner query is independent of the outer query, executes once, and returns its results to the outer query. T-SQL also supports *correlated* subqueries, in which the inner query references column in the outer query and conceptually executes once per row.

Working with Correlated Subqueries

Like self-contained subqueries, correlated subqueries are SELECT statements nested within an outer query. Correlated subqueries may also be either scalar or multi-valued subqueries. They're typically used when the inner query needs to reference a value in the outer query.

However, unlike self-contained subqueries, there are some special considerations when using correlated subqueries:

- Correlated subqueries cannot be executed separately from the outer query. This restriction complicates testing and debugging.
- Unlike self-contained subqueries, which are processed once, correlated subqueries will run multiple times. Logically, the outer query runs first, and for each row returned, the inner query is processed.

The following example uses a correlated subquery to return the most recent order for each customer. The subquery refers to the outer query and references its **CustomerID** value in its WHERE clause. For each row in the outer query, the subquery finds the maximum order ID for the customer referenced in that row, and the outer query checks to see if the row it's looking at is the row with that order ID.

```
SELECT SalesOrderID, CustomerID, OrderDate
FROM SalesLT.SalesOrderHeader AS o1
WHERE SalesOrderID =
    (SELECT MAX(SalesOrderID)
     FROM SalesLT.SalesOrderHeader AS o2
     WHERE o2.CustomerID = o1.CustomerID)
ORDER BY CustomerID, OrderDate;
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Writing Correlated Subqueries

To write correlated subqueries, consider the following guidelines:

- Write the outer query to accept the appropriate return result from the inner query. If the inner query is scalar, you can use equality and comparison operators, such as =, <, >, and <>, in the WHERE clause. If the inner query might return multiple values, use an IN predicate. Plan to handle NULL results.
- Identify the column from the outer query that will be referenced by the correlated subquery. Declare an alias for the table that is the source of the column in the outer query.
- Identify the column from the inner table that will be compared to the column from the outer table. Create an alias for the source table, as you did for the outer query.
- Write the inner query to retrieve values from its source, based on the input value from the outer query. For example, use the outer column in the WHERE clause of the inner query.

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

Working with EXISTS

In addition to retrieving values from a subquery, T-SQL provides a mechanism for checking whether any results would be returned from a query. The EXISTS predicate determines whether any rows meeting a specified condition exist, but rather than return them, it returns TRUE or FALSE. This technique is useful for validating data without incurring the overhead of retrieving and processing the results.

When a subquery is related to the outer query using the EXISTS predicate, SQL Server handles the results of the subquery in a special way. Rather than retrieve a scalar value or a multi-valued list from the subquery, EXISTS simply checks to see if there are any rows in the result.

Conceptually, an EXISTS predicate is equivalent to retrieving the results, counting the rows returned, and comparing the count to zero. Compare the following queries, which will return details about customers who have placed orders:

The first example query uses COUNT in a subquery:

```
SELECT CustomerID, CompanyName, EmailAddress
FROM Sales.Customer AS c
WHERE
(SELECT COUNT(*)
  FROM Sales.SalesOrderHeader AS o
  WHERE o.CustomerID = c.CustomerID) > 0;
```

The second query, which returns the same results, uses EXISTS:

```
SELECT CustomerID, CompanyName, EmailAddress
FROM Sales.Customer AS c
WHERE EXISTS
(SELECT *
  FROM Sales.SalesOrderHeader AS o
  WHERE o.CustomerID = c.CustomerID);
```

In the first example, the subquery must count every occurrence of each *custid* found in the **Sales.Orders** table, and compare the count results to zero, simply to indicate that the customer has placed orders.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

In the second query, EXISTS returns TRUE for a **custid** as soon as a relevant order has been found in the **Sales.Orders** table. A complete accounting of each occurrence is unnecessary. Also note that with the EXISTS form, the subquery is not restricted to returning a single column. Here, we have SELECT *. The returned columns are irrelevant because we're only checking if any rows are returned at all, not what values are in those rows.

Note: From the perspective of logical processing, the two query forms are equivalent. From a performance perspective, the database engine may treat the queries differently as it optimizes them for execution. Consider testing each one for your own usage.

Note: If you're converting a subquery using COUNT(*) to one using EXISTS, make sure the subquery uses a SELECT * and not SELECT COUNT(*). SELECT COUNT(*) **always** returns a row, so the EXISTS will always return **TRUE.**

Another useful application of EXISTS is negating the subquery with NOT, as in the following example, which will return any customer who has never placed an order:

```
SELECT CustomerID, CompanyName, EmailAddress
FROM SalesLT.Customer AS c
WHERE NOT EXISTS
  (SELECT *
   FROM SalesLT.SalesOrderHeader AS o
   WHERE o.CustomerID = c.CustomerID);
```

SQL Server won't have to return data about the related orders for customers who have placed orders. If a **custid** is found in the **Sales.Orders** table, NOT EXISTS evaluates to FALSE and the evaluation quickly completes.

To write queries that use EXISTS with subqueries, consider the following guidelines:

- The keyword EXISTS directly follows WHERE. No column name (or other expression) precedes it, unless NOT is also used.
- Within the subquery, use SELECT *. No rows are returned by the subquery, so no columns need to be specified.

Lab - Use Subqueries

Estimated time: 30 minutes

Depending on the lab hosting solution used for your class delivery, the lab instructions may display directly in the virtual lab environment. In addition, they are available **online in GitHub**[2]. Follow guidance from your instructor to access the lab environment.

Module-Review

Module Review

Use the questions below to check your knowledge of the topics covered in this module.

2 https://microsoftlearning.github.io/dp-080-Transact-SQL/

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Multiple choice

You must return a list of all sales employees that have taken sales orders. Employees who have not taken sales orders should not be included in the results. Which type of join is required?

- ☐ INNER
- ☐ LEFT OUTER
- ☐ FULL OUTER

Multiple choice

You write the following query: SELECT p.Name, c.Name FROM Store.Product AS p CROSS JOIN Store. Category AS c; What does the query return?

- ☐ Only data rows where the product name is the same as the category name
- ☐ Only rows where the product name is not the same as the category name
- ☐ Every combination of product and category name

Multiple choice

Which of the following statements is true about correlated subqueries?

- ☐ A correlated subquery returns a single scalar value
- ☐ A correlated subquery returns multiple columns and rows
- ☐ A correlated subquery references a value in the outer query

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Answers

Multiple choice

You must return a list of all sales employees that have taken sales orders. Employees who have not taken sales orders should not be included in the results. Which type of join is required?

■ INNER

☐ LEFT OUTER

☐ FULL OUTER

Explanation
Use an INNER join to retrieve only records that match in both tables.

Multiple choice

You write the following query: SELECT p.Name, c.Name FROM Store.Product AS p CROSS JOIN Store.Category AS c; What does the query return?

☐ Only data rows where the product name is the same as the category name

☐ Only rows where the product name is not the same as the category name

■ Every combination of product and category name

Explanation
A CROSS JOIN returns every combination from both tables.

Multiple choice

Which of the following statements is true about correlated subqueries?

☐ A correlated subquery returns a single scalar value

☐ A correlated subquery returns multiple columns and rows

■ A correlated subquery references a value in the outer query

Explanation
A correlated subquery references a value in the outer query.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Module 4 Using Built-in Functions

Getting Started with Scalar Functions

Introduction to Built-In Functions

Transact-SQL includes many built-in functions, ranging from functions that perform data type conversion, to functions that aggregate and analyze groups of rows.

Functions in T-SQL can be categorized as follows:

Function Category	Description
Scalar	Operate on a single row, return a single value.
Logical	Compare multiple values to determine a single output.
Ranking	Operate on a partition (set) of rows.
Rowset	Return a virtual table that can be used in a FROM clause in a T-SQL statement.
Aggregate	Take one or more input values, return a single summarizing value.

Scalar Functions

Scalar functions return a single value and usually work on a single row of data. The number of input values they take can be zero (for example, GETDATE), one (for example, UPPER), or multiple (for example, ROUND). As scalar functions always return a single value, they can be used anywhere a single value (the result) could exist in its own right. They are most commonly used in SELECT clauses and WHERE clause predicates. They can also be used in the SET clause of an UPDATE statement.

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

Some considerations when using scalar functions include:

- **Determinism**: If the function returns the same value for the same input and database state each time it is called, we say it is *deterministic*. For example, ROUND(1.1, 0) always returns the value 1.0. Many

built-in functions are *nondeterministic*. For example, GETDATE() returns the current date and time. Results from nondeterministic functions cannot be indexed, which affects the query processor's ability to come up with a good plan for executing the query.

- **Collation**: When using functions that manipulate character data, which collation will be used? Some functions use the collation (sort order) of the input value; others use the collation of the database if no input collation is supplied.

Scalar function examples

At the time of writing, the SQL Server Technical Documentation listed more than 200 scalar functions that span multiple categories, including:

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

There isn't enough time in this course to describe each function, but the examples below show some commonly used functions.

The following hypothetical example uses a number of data and time functions:

```
SELECT  SalesOrderID,
    OrderDate,
        YEAR(OrderDate) AS OrderYear,
        DATENAME(mm, OrderDate) AS OrderMonth,
        DAY(OrderDate) AS OrderDay,
        DATENAME(dw, OrderDate) AS OrderWeekDay,
        DATEDIFF(yy,OrderDate, GETDATE()) AS YearsSinceOrder
FROM Sales.SalesOrderHeader;
```

Partial results are shown below:

SalesOrderID	OrderDate	OrderYear	OrderMonth	OrderDay	OrderWeek-Day	YearsSince-Order
71774	2008-06-01T00:00:00	2008	June	1	Sunday	13
...	...	...	...	...	...	...

The next example includes some mathematical functions:

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

```
SELECT TaxAmt,
       ROUND(TaxAmt, 0) AS Rounded,
       FLOOR(TaxAmt) AS Floor,
       CEILING(TaxAmt) AS Ceiling,
       SQUARE(TaxAmt) AS Squared,
       SQRT(TaxAmt) AS Root,
       LOG(TaxAmt) AS Log,
       TaxAmt * RAND() AS Randomized
FROM Sales.SalesOrderHeader;
```

Partial results:

TaxAmt	Rounded	Floor	Ceiling	Squared	Root	Log	Rand-omized
70.4279	70.0000	70.0000	71.0000				
…	..	…	…	…	…	…	…

The following example uses some text functions:

```
SELECT  CompanyName,
        UPPER(CompanyName) AS UpperCase,
        LOWER(CompanyName) AS LowerCase,
        LEN(CompanyName) AS Length,
        REVERSE(CompanyName) AS Reversed,
        CHARINDEX(' ', CompanyName) AS FirstSpace,
        LEFT(CompanyName, CHARINDEX(' ', CompanyName)) AS FirstWord,
        SUBSTRING(CompanyName, CHARINDEX(' ', CompanyName) + 1, LEN(Compa-
nyName)) AS RestOfName
FROM Sales.Customer;
```

Partial results:

Compa-nyName	UpperCase	LowerCase	Length	Reversed	FirstSpace	FirstWord	RestOf-Name
A Bike Store	A BIKE STORE	a bike store	12	erotS ekiB A	2	A	Bike Store
Progres-sive Sports	PROGRES-SIVE SPORTS	progres-sive sports	18	stropS evisser-gorP	12	Progres-sive	Sports
Advanced Bike Compo-nents	AD-VANCED BIKE COMPO-NENTS	advanced bike compo-nents	24	stnenop-moC ekiB decnavdA	9	Advanced	Bike Compo-nents
…	…	…	…	…	…	…	…

Logical Functions

Logical functions evaluate an input expression, and return an appropriate value based on the result.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

IIF

The **IIF** function evaluates a Boolean input expression, and returns a specified value if the expression evaluates to **True**, and an alternative value if the expression evaluates to **False**.

For example, consider the following query, which evaluates the address type of a customer. If the value is "Main Office", the expression returns "Billing". For all other address type values, the expression returns "Mailing".

```
SELECT AddressType,
       IIF(AddressType = 'Main Office', 'Billing', 'Mailing') AS UseAddress-
For
FROM Sales.CustomerAddress;
```

The partial results from this query might look like this:

AddressType	UseAddressFor
Main Office	Billing
Shipping	Mailing
...	...

CHOOSE

The **CHOOSE** function evaluates an integer expression, and returns the corresponding value from a list based on its (1-based) ordinal position.

```
SELECT SalesOrderID, Status,
CHOOSE(Status, 'Ordered', 'Shipped', 'Delivered') AS OrderStatus
FROM Sales.SalesOrderHeader;
```

The results from this query might look something like this:

SalesOrderID	Status	OrderStatus
1234	3	Delivered
1235	2	Shipped
1236	2	Shipped
1237	1	Ordered
...	...	...

Ranking Functions

Ranking functions allow you to perform calculations against a user-defined set of rows. These functions include ranking, offset, aggregate, and distribution functions.

This example uses the RANK function to calculate a ranking based on the **ListPrice**, with the highest price ranked at 1:

```
SELECT TOP 100 ProductID, Name, ListPrice,
RANK() OVER(ORDER BY ListPrice DESC) AS RankByPrice
FROM Production.Product AS p
ORDER BY RankByPrice;
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

The results of this query might look similar to the following:

ProductID	Name	ListPrice	RankByPrice
749	Road-150 Red, 62	3578.27	1
750	Road-150 Red, 44	3578.27	1
751	Road-150 Red, 48	3578.27	1
771	Mountain-100 Silver, 38	3399.99	4
772	Mountain-100 Silver, 42	3399.99	4
775	Mountain-100 Black, 38	3374.99	6
...	...	...	...

You can use the OVER clause to define partitions, or groupings within the data. For example, the following query extends the previous example to calculate price-based rankings for products within each category.

```
SELECT c.Name AS Category, p.Name AS Product, ListPrice,
  RANK() OVER(PARTITION BY c.Name ORDER BY ListPrice DESC) AS RankByPrice
FROM Production.Product AS p
JOIN Production.ProductCategory AS c
ON p.ProductCategoryID = c.ProductcategoryID
ORDER BY Category, RankByPrice;
```

The results of this query might look something like this:

Category	Product	ListPrice	RankByPrice
Bib-Shorts	Men's Bib-Shorts, S	89.99	1
Bib-Shorts	Men's Bib-Shorts, M	89.99	1
Bike Racks	Hitch Rack - 4-Bike	120	1
Bike Stands	All-Purpose Bike Stand	159	1
Bottles and Cages	Mountain Bottle Cage	9.99	1
Bottles and Cages	Road Bottle Cage	8.99	2
Bottles and Cages	Water Bottle - 30 oz.	4.99	3
Bottom Brackets	HL Bottom Bracket	121.49	1
Bottom Brackets	ML Bottom Bracket	101.24	2
Bottom Brackets	LL Bottom Bracket	53.99	3
...	...	...	...

Rowset Functions

Rowset functions return a virtual table that can be used in the FROM clause as a data source. These functions take parameters specific to the rowset function itself. They include OPENDATASOURCE, OPENQUERY, OPENROWSET, OPENXML, and OPENJSON.

The OPENDATASOURCE, OPENQUERY, and OPENROWSET functions enable you to pass a query to a remote database server. The remote server will then return a set of result rows. For example, the following query uses OPENROWSET to get the results of a query from a SQL Server instance named **SalesDB**.

```
SELECT a.*
FROM OPENROWSET('SQLNCLI', 'Server=SalesDB;Trusted_Connection=yes;',
     'SELECT Name, ListPrice
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

```
        FROM AdventureWorks.Production.Product') AS a;
```

Note: To use remote servers, you must enable some advanced options in the SQL Server instance where you're running the query.

The OPENXML and OPENJSON functions enable you to query structured data in XML or JSON format and extract values into a tabular rowset.

Note: A detailed exploration of rowset functions is beyond the scope of this course. See the **Transact-SQL reference documentation**[1] for more information.

Aggregate Functions

T-SQL provides aggregate functions such as SUM, MAX, and AVG to perform calculations that take multiple values and return a single result.

Working with Aggregate Functions

Most of the queries we have looked at operate on a row at a time, using a WHERE clause to filter rows. Each row returned corresponds to one row in the original data set.

Many aggregate functions are provided in SQL Server. In this section, we'll look at the most common functions such as SUM, MIN, MAX, AVG, and COUNT.

When working with aggregate functions, you need to consider the following points:

- Aggregate functions return a single (scalar) value and can be used in SELECT statements almost anywhere a single value can be used. For example, these functions can be used in the SELECT, HAVING, and ORDER BY clauses. However, they cannot be used in the WHERE clause.
- Aggregate functions ignore NULLs, except when using COUNT(*).
- Aggregate functions in a SELECT list don't have a column header unless you provide an alias using AS.
- Aggregate functions in a SELECT list operate on all rows passed to the SELECT operation. If there is no GROUP BY clause, all rows satisfying any filter in the WHERE clause will be summarized. You will learn more about GROUP BY in the next topic.
- Unless you're using GROUP BY, you shouldn't combine aggregate functions with columns not included in functions in the same SELECT list.

To extend beyond the built-in functions, SQL Server provides a mechanism for user-defined aggregate functions via the .NET Common Language Runtime (CLR). That topic is beyond the scope of this module.

Built-in Aggregate Functions

As mentioned, Transact-SQL provides many built-in aggregate functions. Commonly used functions include:

Function Name	Syntax	Description
SUM	SUM(*expression*)	Totals all the non-NULL numeric values in a column.

1 https://docs.microsoft.com/sql/t-sql/functions/functions

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Function Name	Syntax	Description
AVG	AVG(*expression*)	Averages all the non-NULL numeric values in a column (sum/count).
MIN	MIN(*expression*)	Returns the largest number, earliest date/time, or first-occurring string (according to collation sort rules).
MAX	MAX(*expression*)	Returns the largest number, latest date/time, or last-occurring string (according to collation sort rules).
COUNT or COUNT_BIG	COUNT(*) or COUNT(*expression*)	With (*), counts all rows, including rows with NULL values. When a column is specified as *expression*, returns the count of non-NULL rows for that column. COUNT returns an int; COUNT_BIG returns a big_int.

To use a built-in aggregate in a SELECT clause, consider the following example in the *MyStore* sample database:

```
SELECT AVG(ListPrice) AS AveragePrice,
       MIN(ListPrice) AS MinimumPrice,
       MAX(ListPrice) AS MaximumPrice
FROM Production.Product;
```

The results of this query look something like this:

AveragePrice	MinimumPrice	MaximumPrice
744.5952	2.2900	3578.2700

Note that the above example summarizes all rows from the **Production.Product** table. We could easily modify the query to return the average, minimum, and maximum prices for products in a specific category by adding a WHERE clause, like this:

```
SELECT AVG(ListPrice) AS AveragePrice,
       MIN(ListPrice) AS MinimumPrice,
       MAX(ListPrice) AS MaximumPrice
FROM Production.Product
WHERE ProductCategoryID = 15;
```

When using aggregates in a SELECT clause, all columns referenced in the SELECT list must be used as inputs for an aggregate function, or be referenced in a GROUP BY clause.

Consider the following query, which attempts to include the **ProductCategoryID** field in the aggregated results:

```
SELECT ProductCategoryID, AVG(ListPrice) AS AveragePrice,
MIN(ListPrice) AS MinimumPrice,
MAX(ListPrice) AS MaximumPrice
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

```
FROM Production.Product;
```

Running this query results in the following error

Msg 8120, Level 16, State 1, Line 1

Column 'Production.ProductCategoryID' is invalid in the select list because it isn't contained in either an aggregate function or the GROUP BY clause.

The query treats all rows as a single aggregated group. Therefore, all columns must be used as inputs to aggregate functions.

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

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

```
SELECT MIN(CompanyName) AS MinCustomer,
       MAX(CompanyName) AS MaxCustomer
FROM SalesLT.Customer;
```

This query will return the first and last values for **CompanyName** in the database's collation sequence, which in this case is alphabetical order:

MinCustomer	MaxCustomer
A Bike Store	Yellow Bicycle Company

Other functions may be nested with aggregate functions.

For example, the YEAR scalar function is used in the following example to return only the year portion of the order date, before MIN and MAX are evaluated:

```
SELECT MIN(YEAR(OrderDate)) AS Earliest,
       MAX(YEAR(OrderDate)) AS Latest
FROM Sales.SalesOrderHeader;
```

Earliest	Latest
2008	2021

The MIN and MAX functions can also be used with date data, to return the earliest and latest chronological values. However, AVG and SUM can only be used for numeric data, which includes integers, money, float and decimal datatypes.

Using DISTINCT with Aggregate Functions

You should be aware of the use of DISTINCT in a SELECT clause to remove duplicate rows. When used with an aggregate function, DISTINCT removes duplicate values from the input column before computing the summary value. DISTINCT is useful when summarizing unique occurrences of values, such as customers in the orders table.

The following example returns the number of customers who have placed orders, no matter how many orders they placed:

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

```
SELECT COUNT(DISTINCT CustomerID) AS UniqueCustomers
FROM Sales.SalesOrderHeader;
```

COUNT(<some_column>) merely counts how many rows have some value in the column. If there are no NULL values, COUNT(<some_column>) will be the same as COUNT(*). COUNT (DISTINCT <some_column>) counts how many different values there are in the column.

Using Aggregate Functions with NULL

It is important to be aware of the possible presence of NULLs in your data, and of how NULL interacts with T-SQL query components, including aggregate function. There are a few considerations to be aware of:

- With the exception of COUNT used with the (*) option, T-SQL aggregate functions ignore NULLs. For example, a SUM function will add only non-NULL values. NULLs don't evaluate to zero. COUNT(*) counts all rows, regardless of value or non-value in any column.
- The presence of NULLs in a column may lead to inaccurate computations for AVG, which will sum only populated rows and divide that sum by the number of non-NULL rows. There may be a difference in results between AVG(<column>) and (SUM(<column>)/COUNT(*)).

For example, consider the following table named t1:

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

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

```
SELECT SUM(c2) AS sum_nonnulls,
    COUNT(*) AS count_all_rows,
    COUNT(c2) AS count_nonnulls,
    AVG(c2) AS average,
    (SUM(c2)/COUNT(*)) AS arith_average
FROM t1;
```

The result would be:

sum_nonnulls	count_all_rows	count_nonnulls	average	arith_average
150	6	5	30	25

In this resultset, the column named **average** is the aggregate that internally gets the sum of 150 and divides by the count of non-null values in column **c2**. The calculation would be 150/5, or 30. The column called **arith_average** explicitly divides the sum by the count of all rows, so the calculation is 150/6, or 25.

If you need to summarize all rows, whether NULL or not, consider replacing the NULLs with another value that will not be ignored by your aggregate function. You can use the COALESCE function for this purpose.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Grouping Aggregated Results

Grouping with GROUP BY

While aggregate functions are useful for analysis, you may wish to arrange your data into subsets before summarizing it. In this section, you will learn how to accomplish this using the GROUP BY clause.

Using the GROUP BY Clause

As you've learned, when your SELECT statement is processed, after the FROM clause and WHERE clause have been evaluated, a virtual table is created. The contents of the virtual table are now available for further processing. You can use the GROUP BY clause to subdivide the contents of this virtual table into groups of rows.

To group rows, specify one or more elements in the GROUP BY clause:

```
GROUP BY <value1> [, <value2>, …]
```

GROUP BY creates groups and places rows into each group as determined by the elements specified in the clause.

For example, the following query will result in a set of grouped rows, one row per **CustomerID** in the **Sales.SalesOrderHeader** table. Another way of looking at the GROUP BY process, is that all rows with the same value for **CustomerID** will be grouped together and returned in a single result row.

```
SELECT CustomerID
FROM Sales.SalesOrderHeader
GROUP BY CustomerID;
```

The query above is equivalent to the following query:

```
SELECT DISTINCT CustomerID
FROM Sales.SalesOrderHeader
```

Once the GROUP BY clause has been processed and each row has been associated with a group, later phases of the query must aggregate any elements of the source rows that are in the SELECT list but that don't appear in the GROUP BY list. This requirement will have an impact on how you write your SELECT and HAVING clauses.

So, what's the difference between writing the query with a GROUP BY or a DISTINCT? If all you want to know is the distinct values for **CustomerID**, there is no difference. But with GROUP BY, we can add other elements to the SELECT list that are then aggregated for each group.

The simplest aggregate function is COUNT(*). The following query takes the original 830 source rows from **CustomerID** and groups them into 89 groups, based on the **CustomerID** values. Each distinct **CustomerID** value generates one row of output in the GROUP BY query

```
SELECT CustomerID, COUNT(*) AS OrderCount
FROM Sales.SalesOrderHeader
GROUP BY CustomerID;
```

For each **CustomerID** value, the query aggregates and counts the rows, so we result shows us how many rows in the **SalesOrderHeader** table belong to each customer.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

CustomerID	OrderCount
1234	3
1005	1

Note that GROUP BY does not guarantee the order of the results. Often, as a result of the way the grouping operation is performed by the query processor, the results are returned in the order of the group values. However, you should not rely on this behavior. If you need the results to be sorted, you must explicitly include an ORDER clause:

```
SELECT CustomerID, COUNT(*) AS OrderCount
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
ORDER BY CustomerID;
```

This time, the results are returned in the specified order:

CustomerID	OrderCount
1005	1
1234	3

The clauses in a SELECT statement are applied in the following order:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

Column aliases are assigned in the SELECT clause, which occurs *after* the GROUP BY clause but *before* the ORDER BY clause. You can reference a column alias in the ORDER BY clause, but not in the GROUP BY clause. The following query will result in an i*nvalid column name* error:

```
SELECT CustomerID AS Customer,
       COUNT(*) AS OrderCount
FROM Sales.SalesOrderHeader
GROUP BY Customer
ORDER BY Customer;
```

However, the following query will succeed, grouping and sorting the results by the customer ID.

```
SELECT CustomerID AS Customer,
       COUNT(*) AS OrderCount
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
ORDER BY Customer;
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Troubleshooting GROUP BY errors

A common obstacle to becoming comfortable with using GROUP BY in SELECT statements is understanding why the following type of error message occurs:

Msg 8120, Level 16, State 1, Line 2 Column <column_name> is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.

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

```
SELECT CustomerID, COUNT(*) AS OrderCount
FROM Sales.SalesOrderHeader
GROUP BY CustomerID;
```

The following query will return an error because **PurchaseOrderNumber** isn't part of the GROUP BY, and it isn't used with an aggregate function.

```
SELECT CustomerID, PurchaseOrderNumber, COUNT(*) AS OrderCount
FROM Sales.SalesOrderHeader
GROUP BY CustomerID;
```

This query returns the error:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Sales.SalesOrderHeader.PurchaseOrderNumber' is invalid in the
select list because it is not contained in either an aggregate function or
the GROUP BY clause.
```

Here's another way to think about it. This query returns one row for each **CustomerID** value. But rows for the same **CustomerID** can have different **PurchaseOrderNumber** values, so which of the values is the one that should be returned?

If you want to see orders per customer ID and per purchase order, you can add the **PurchaseOrderNumber** column to the GROUP BY clause, as follows:

```
SELECT CustomerID, OrderDate, COUNT(*) AS OrderCount
FROM Sales.SalesOrderHeader
GROUP BY CustomerID, PurchaseOrderNumber;
```

This query will return one row for each customer and each purchase order combination, along with the count of orders for that combination.

Filtering Groups with HAVING

When you have created groups with a GROUP BY clause, you can further filter the results. The HAVING clause acts as a filter on groups. This is similar to the way that the WHERE clause acts as a filter on rows returned by the FROM clause.

A HAVING clause enables you to create a search condition, conceptually similar to the predicate of a WHERE clause, which then tests each group returned by the GROUP BY clause.

The following example counts the orders for each customer, and filters the results to include only customers that have placed more than 10 orders:

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

```
SELECT CustomerID,
      COUNT(*) AS OrderCount
FROM Sales.SalesOrderHeader
GROUP BY CustomerID
HAVING COUNT(*) > 10;
```

Compare HAVING to WHERE

While both HAVING and WHERE clauses filter data, remember that WHERE operates on rows returned by the FROM clause. If a GROUP BY ... HAVING section exists in your query following a WHERE clause, the WHERE clause will filter rows before GROUP BY is processed—potentially limiting the groups that can be created.

A HAVING clause is processed after GROUP BY and only operates on groups, not detail rows. To summarize:

- A WHERE clause filters rows before any groups are formed
- A HAVING clause filters entire groups, and usually looks at the results of an aggregation.

Lab - Use Built-In Functions

Estimated time: 30 minutes

Depending on the lab hosting solution used for your class delivery, the lab instructions may display directly in the virtual lab environment. In addition, they are available **online in GitHub**[2]. Follow guidance from your instructor to access the lab environment.

Module-Review

Module Review

Use the questions below to check your knowledge of the topics covered in this module.

Multiple choice

You run the following query: SELECT OrderNo, CHOOSE(Status, 'Ordered', 'Shipped', 'Delivered') AS OrderState FROM Sales.Order; Which OrderState value is returned for rows with a Status value of 2?

- ☐ Shipped
- ☐ Delivered
- ☐ NULL

2 https://microsoftlearning.github.io/dp-080-Transact-SQL/

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Multiple choice

You must return the number of customers in each city. Which query should you use?

- ☐ SELECT City, COUNT(*) AS CustomerCount FROM Sales.Customer;
- ☐ SELECT City, COUNT(*) AS CustomerCount FROM Sales.Customer GROUP BY City;
- ☐ SELECT City, COUNT(*) AS CustomerCount FROM Sales.Customer ORDER BY City;

Multiple choice

You must return a row for each category with an average price over 10.00. Which query should you use?

- ☐ SELECT Category, AVG(Price) FROM Store.Product WHERE AVG(Price) > 10.00;
- ☐ SELECT Category, AVG(Price) FROM Store.Product GROUP BY Category WHERE AVG(Price) > 10.00;
- ☐ SELECT Category, AVG(Price) FROM Store.Product GROUP BY Category HAVING AVG(Price) > 10.00;

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Answers

Multiple choice

You run the following query: SELECT OrderNo, CHOOSE(Status, 'Ordered', 'Shipped', 'Delivered') AS OrderState FROM Sales.Order; Which OrderState value is returned for rows with a Status value of 2?

- ■ Shipped
- ☐ Delivered
- ☐ NULL

Explanation
CHOOSE returns the value based on its 1-based ordinal position.

Multiple choice

You must return the number of customers in each city. Which query should you use?

- ☐ SELECT City, COUNT(*) AS CustomerCount FROM Sales.Customer;
- ■ SELECT City, COUNT(*) AS CustomerCount FROM Sales.Customer GROUP BY City;
- ☐ SELECT City, COUNT(*) AS CustomerCount FROM Sales.Customer ORDER BY City;

Explanation
Use GROUP by to aggregate by groups.

Multiple choice

You must return a row for each category with an average price over 10.00. Which query should you use?

- ☐ SELECT Category, AVG(Price) FROM Store.Product WHERE AVG(Price) > 10.00;
- ☐ SELECT Category, AVG(Price) FROM Store.Product GROUP BY Category WHERE AVG(Price) > 10.00;
- ■ SELECT Category, AVG(Price) FROM Store.Product GROUP BY Category HAVING AVG(Price) > 10.00;

Explanation
Use a HAVING clause to filter groups.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Module 5 Modifying Data

Inserting Data into Tables

Options for Inserting Data

Transact-SQL provides multiple ways to insert rows into a table.

The INSERT statement

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

The basic syntax of a simple INSERT statement is shown below:

```
INSERT [INTO] <Table> [(column_list)]
VALUES ([ColumnName or an expression or DEFAULT or NULL],…n)
```

With this form of the INSERT statement, called INSERT VALUES, you can specify the columns that will have values placed in them and the order in which the data will be presented for each row inserted into the table. The column_list is optional but recommended. Without the column_list, the INSERT statement will expect a value for every column in the table in the order in which the columns were defined. You can also provide the values for those columns as a comma-separated list.

When listing values, the keyword DEFAULT means a predefined value, that was specified when the table was created, will be used. There are three ways a default can be determined:

- If a column has been defined to have an automatically generated value, that value will be used. Autogenerated values will be discussed later in this module.
- When a table is created, a default value can be supplied for a column, and that value will be used if DEFAULT is specified.
- If a column has been defined to allow NULL values, and the column isn't an autogenerated column and doesn't have a default defined, NULL will be inserted as a DEFAULT.

NOTE: The details of table creation are beyond the scope of this module. However, it is often useful to see what columns are in a table. The easiest way is to just execute a SELECT statement on the table

without returning any rows. By using a WHERE condition that can never be TRUE, no rows can be returned.

```
SELECT * FROM Sales.Promotion
WHERE 1 = 0;
```

This statement will show you all the columns and their names, but won't show the data types or any properties, such as whether NULLs are allowed, or if there is a default values specified. An example of the output from the query might look like this:

PromotionName	StartDate	ProductModelID	Discount	Notes

To insert data into this table, you can use the INSERT statement as shown here.

```
INSERT INTO Sales.Promotion (PromotionName,StartDate,ProductModelID,Dis-
count,Notes)
VALUES
('Clearance Sale', '01/01/2021', 23, 0.1, '10% discount');
```

For this example above, the column list can be omitted, because we're supplying a value for every column in the correct order:

```
INSERT INTO Sales.Promotion
VALUES
('Clearance Sale', '01/01/2021', 23, 0.1, '10% discount');
```

Suppose that the table is defined such that a default value of the current date is applied to the **StartDate** column, and the **Notes** column allows NULL values. You can indicate that you want to use these values explicitly, like this:

```
INSERT INTO Sales.Promotion
VALUES
('Pull your socks up', DEFAULT, 24, 0.25, NULL);
```

Alternatively, you can omit values in the INSERT statement, in which case the default value will be used if defined, and if there is no default value but the column allows NULLs, then a NULL will be inserted. If you're not supplying values for all columns, you must have a column list indicated which column values you're supplying.

```
INSERT INTO Sales.Promotion (PromotionName, ProductModelID, Discount)
VALUES
('Caps Locked', 2, 0.2);
```

In addition to inserting a single row at a time, the INSERT VALUES statement can be used to insert multiple rows by providing multiple comma-separated sets of values. The sets of values are also separated by commas, like this:

```
(col1_val,col2_val,col3_val),
(col1_val,col2_val,col3_val)
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

This list of values is known as a **table value constructor**. Here's an example of inserting two more rows into our table with a table value constructor:

```
INSERT INTO Sales.Promotion
VALUES
('The gloves are off!', DEFAULT, 3, 0.25, NULL),
('The gloves are off!', DEFAULT, 4, 0.25, NULL);
```

INSERT ... SELECT

In addition to specifying a literal set of values in an INSERT statement, T-SQL also supports using the results of other operations to provide values for INSERT. You can use the results of a SELECT statement or the output of a stored procedure to supply the values for the INSERT statement.

To use the INSERT with a nested SELECT, build a SELECT statement to replace the VALUES clause. With this form, called INSERT SELECT, you can insert the set of rows returned by a SELECT query into a destination table. The use of INSERT SELECT presents the same considerations as INSERT VALUES:

- You may optionally specify a column list following the table name.
- You must provide column values or DEFAULT, or NULL, for each column.

The following syntax illustrates the use of INSERT SELECT:

```
INSERT [INTO] <table or view> [(column_list)]
SELECT <column_list> FROM <table_list>...;
```

Note: Result sets from stored procedures (or even dynamic batches) may also be used as input to an INSERT statement. This form of INSERT, called INSERT EXEC, is conceptually similar to INSERT SELECT and will present the same considerations. However, stored procedures can return multiple result sets, so extra care is needed.

The following example inserts multiple rows for a new promotion named *Get Framed* by retrieving the model ID and model name from the **Production.ProductModel**, table for every model that contains "frame" in its name.

```
INSERT INTO Sales.Promotion (PromotionName, ProductModelID, Discount,
Notes)
SELECT DISTINCT 'Get Framed', m.ProductModelID, 0.1, '10% off ' + m.Name
FROM Production.ProductModel AS m
WHERE m.Name LIKE '%frame%';
```

Unlike a subquery, the nested SELECT used with an INSERT isn't enclosed in parentheses.

SELECT ... INTO

Another option for inserting rows, which is similar to INSERT SELECT, is the SELECT INTO statement. The biggest difference between INSERT SELECT and SELECT INTO is that SELECT INTO cannot be used to insert rows into an existing table, because it always creates a new table that is based on the result of the SELECT. Each column in the new table will have the same name, data type, and nullability as the corresponding column (or expression) in the SELECT list.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

To use SELECT INTO, add INTO <new_table_name> in the SELECT clause of the query, just before the FROM clause. Here's an example that extracts data from the **Sales.SalesOrderHeader** table into a new table named **Sales.Invoice**..

```
SELECT SalesOrderID, CustomerID, OrderDate, PurchaseOrderNumber, TotalDue
INTO Sales.Invoice
FROM Sales.SalesOrderHeader;
```

A SELECT INTO will fail if there already is a table with the name specified after INTO. After the table is created, it can be treated like any other table. You can select from it, join it to other tables, or insert more rows into it.

Identity Columns

You may need to automatically generate sequential values for one column in a specific table. Transact-SQL provides the IDENTITY property.

To use the IDENTITY property, define a column using a numeric data type with a scale of 0 (meaning whole numbers only) and include the IDENTITY keyword. The allowable types include all integer types and decimal types where you explicitly give a scale of 0.

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

Note: The IDENTITY property is specified in place of specifying NOT or NOT NULL in the column definition. Any column with the IDENTITY property is automatically not nullable. You can specify NOT NULL just for self-documentation, but if you specify the column as NULL (meaning nullable), the table creation statement will generate an error.

Only one column in a table may have the IDENTITY property set; it's frequently used as either the PRIMARY KEY or an alternate key.

The following code shows the creation of the **Sales.Promotion** table used in the previous section examples, but this time with an identity column named **PromotionID** as the primary key:

```
CREATE TABLE Sales.Promotion
(
PromotionID int IDENTITY PRIMARY KEY,
PromotionName varchar(20),
StartDate datetime NOT NULL DEFAULT GETDATE(),
ProductModelID int NOT NULL REFERENCES Production.ProductModel(ProductMod-
elID),
Discount decimal(4,2) NOT NULL,
Notes nvarchar(max) NULL
);
```

Note: The full details of the CREATE TABLE statement are beyond the scope of this module.

Inserting data into an identity column

When an IDENTITY property is defined for a column, INSERT statements into the table generally don't specify a value for the IDENTITY column. The database engine generates a value using the next available value for the column.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

For example, you could insert a row into the **Sales.Promotion** table without specifying a value for the **PromotionID** column:

```
INSERT INTO Sales.Promotion
VALUES
('Clearance Sale', '01/01/2021', 23, 0.10, '10% discount')
```

Note that even though the VALUES clause doesn't include a value for the **PromotionID** column, you don't need to specify a column list in the INSERT clause - Identity columns are exempt from this requirement.

If this row is the first one inserted into the table, the result is a new row like this:

PromotionID	Promotion-Name	StartDate	ProductMod-elID	Discount	Notes
1	Clearance Sale	2021-01-01T00:00:00	23	0.1	10% discount

When the table was created, no seed or increment values were set for the IDENTITY column, so the first row is inserted with a value of 1. The next row to be inserted will be assigned a **PromotionID** value of 2, and so on.

Retrieving an identity value

To return the most recently assigned IDENTITY value within the same session and scope, use the SCOPE_IDENTITY function; like this:

```
SELECT SCOPE_IDENTITY();
```

The SCOPE_IDENTITY function returns the most recent identity value generated in the current scope for any table. If you need the latest identity value in a specific table, you can use the IDENT_CURRENT function, like this:

```
SELECT IDENT_CURRENT('Sales.Promotion');
```

Overriding identity values

If you want to override the automatically generated value and assign a specific value to the IDENTITY column, you first need to enable identity inserts by using the SET IDENTITY INSERT *table_name* ON statement. With this option enabled, you can insert an explicit value for the identity column, just like any other column. When you're finished, you can use the SET IDENTITY INSERT *table_name* OFF statement to resume using automatic identity values, using the last value you explicitly entered as a seed.

```
SET IDENTITY_INSERT SalesLT.Promotion ON;

INSERT INTO SalesLT.Promotion (PromotionID, PromotionName, ProductModelID,
Discount)
VALUES
(20, 'Another short sale',37, 0.3);
```

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

```
SET IDENTITY_INSERT SalesLT.Promotion OFF;
```

Sequences

As you have learned, the IDENTITY property is used to generate a sequence of values for a column within a table. However, the IDENTITY property is not suitable for coordinating values across multiple tables within a database. For example, suppose your organization differentiates between direct sales and sales to resellers, and wants to store data for these sales in separate tables. Both kinds of sale may need a unique invoice number, and you may want to avoid duplicating the same value for two different kinds of sale. One solution for this requirement is to maintain a pool of unique sequential values across both tables.

In Transact-SQL, you can use a sequence object to provision new sequential values independently of a specific table. A sequence object is created using the CREATE SEQUENCE statement, optionally supplying the data type (must be an integer type or decimal or numeric with a scale of 0), the starting value, an increment value, a maximum value, and other options related to performance.

```
CREATE SEQUENCE Sales.InvoiceNumber AS INT
START WITH 1000 INCREMENT BY 1;
```

To retrieve the next available value from a sequence, use the NEXT VALUE FOR construct, like this:

```
INSERT INTO Sales.ResellerInvoice
VALUES
(NEXT VALUE FOR Sales.InvoiceNumber, 2, GETDATE(), 'PO12345', 107.99);
```

IDENTITY or SEQUENCE

When deciding whether to use IDENTITY columns or a SEQUENCE object for autopopulating values, keep the following points in mind:

- Use SEQUENCE if your application requires sharing a single series of numbers between multiple tables or multiple columns within a table.
- SEQUENCE allows you to sort the values by another column. The NEXT VALUE FOR construct can use the OVER clause to specify the sort column. The OVER clause guarantees that the values returned are generated in the order of the OVER clause's ORDER BY clause. This functionality also allows you to generate row numbers for rows as they're being returned in a SELECT. In the following example, the **Production.Product** table is sorted by the **Name** column, and the first returned column is a sequential number.

  ```
  SELECT NEXT VALUE FOR dbo.Sequence OVER (ORDER BY Name) AS NextID,
    ProductID,
    Name
  FROM Production.Product;
  ```

  Note: Even though the previous statement was just selecting SEQUENCE values to display, the values are still being 'used up' and the displayed SEQUENCE values will no longer be available. If you run the above SELECT multiple times, you'll get different SEQUENCE values each time.

- Use SEQUENCE if your application requires multiple numbers to be assigned at the same time. For example, an application needs to reserve five sequential numbers. Requesting identity values could

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

result in gaps in the series if other processes were simultaneously issued numbers. You can use the **sp_sequence_get_range** system procedure to retrieve several numbers in the sequence at once.

- SEQUENCE allows you to change the specification of the sequence, such as the increment value.
- IDENTITY values are protected from updates. If you try to update a column with the IDENTITY property, you'll get an error.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Modifying and Deleting Data

Updating Data in a Table

The UPDATE statement in T-SQL is used to change existing data in a table. UPDATE operates on a set of rows, either defined by a condition in a WHERE clause or defined in a join. The UPDATE statement has a SET clause that specifies which columns are to be modified. The SET clause one or more columns, separated by commas, and supplies new values to those columns. The WHERE clause in an UPDATE statement has the same structure as a WHERE clause in a SELECT statement.

Note: It's important to note that an UPDATE without a corresponding WHERE clause or a join, will update all the rows in a table. Use the UPDATE statement with caution.

The basic syntax of an UPDATE statement is shown below.

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

The following example shows the UPDATE statement used to modify the notes for a promotion:

```
UPDATE Sales.Promotion
SET Notes = '25% off socks'
WHERE PromotionID = 2;
```

You can modify multiple columns in the SET clause. For example, the following UPDATE statement modified both the **Discount** and **Notes** fields for all rows where the promotion name is "Get Framed":

```
UPDATE Sales.Promotion
SET Discount = 0.2, Notes = REPLACE(Notes, '10%', '20%')
WHERE PromotionName = 'Get Framed';
```

The UPDATE statement also supports a FROM clause, enabling you to modify data based on the results of a query. For example, the following code updates the **Sales.Promotion** table using values retrieved from the **Product.ProductModel** table.

```
UPDATE Sales.Promotion
SET Notes = FORMAT(Discount, 'P') + ' off ' + m.Name
FROM Product.ProductModel AS m
WHERE Notes IS NULL
    AND Sales.Promotion.ProductModelID = m.ProductModelID;
```

Deleting Data From a Table

Just as the INSERT statement always adds whole rows to a table, the DELETE statement always removes entire rows.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Using DELETE to remove specific rows

DELETE operates on a set of rows, either defined by a condition in a WHERE clause or defined in a join. The WHERE clause in a DELETE statement has the same structure as a WHERE clause in a SELECT statement.

Note: It's important to keep in mind that a DELETE without a corresponding WHERE clause will remove all the rows from a table. Use the DELETE statement with caution.

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

```
DELETE [FROM] <TableName>
WHERE <search_conditions>
```

The following example uses the DELETE statement to remove all products from the specified table that have been discontinued. There's a column in the table called *discontinued* and for products that are no longer available, the column has a value of 1.

```
DELETE FROM Production.Product
WHERE discontinued = 1;
```

Using TRUNCATE TABLE to remove all rows

DELETE without a WHERE clause removes all the rows from a table. For this reason, DELETE is usually used conditionally, with a filter in the WHERE clause. If you really do want to remove all the rows and leave an empty table, you can use the TRUNCATE TABLE statement. This statement does not allow a WHERE clause and always removes all the rows in one operation. Here's an example:

```
TRUNCATE TABLE Sales.Sample;
```

TRUNCATE TABLE is more efficient than DELETE when you do want to remove all rows.

Merging Data

In database operations, there is sometimes a need to perform a SQL MERGE operation. This DML option allows you to synchronize two tables by inserting, updating, or deleting rows in one table based on differences found in the other table. The table that is being modified is referred to as the *target* table. The table that is used to determine which rows to change are called the *source* table.

MERGE modifies data, based on one or more conditions:

- When the source data has a matching row in the target table, it can update data in the target table.
- When the source data has no match in the target, it can insert data into the target table.
- When the target data has no match in the source, it can delete the target data.

The general syntax of a MERGE statement is shown below. We're matching the target and the source on a specified column, and if there's a match between target and source, we specify an action to take on the target table. If there's not a match, we specify an action. The action can be an INSERT, UPDATE, or DELETE operation. This code indicates that an UPDATE is performed when there's a match between the source and the target. An INSERT is performed when there's data in the source with no matching data in the target. Finally, a DELETE is performed when there is data in the target with no match in the source. There are many other possible forms of a MERGE statement.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

```
MERGE INTO schema_name.table_name AS TargetTbl
USING (SELECT <select_list>) AS SourceTbl
ON (TargetTbl.col1 = SourceTbl.col1)
WHEN MATCHED THEN
   UPDATE SET TargetTbl.col2 = SourceTbl.col2
WHEN NOT MATCHED [BY TARGET] THEN
   INSERT (<column_list>)
   VALUES (<value_list>)
WHEN NOT MATCHED BY SOURCE THEN
   DELETE;
```

You can use only the elements of the MERGE statement that you need. For example, suppose the database includes a table of staged invoice updates, that includes a mix of revisions to existing invoices and new invoices. You can use the WHEN MATCHED and WHEN NOT MATCHED clauses to update or insert invoice data as required.

```
MERGE INTO Sales.Invoice as i
USING Sales.InvoiceStaging as s
ON i.SalesOrderID = s.SalesOrderID
WHEN MATCHED THEN
    UPDATE SET i.CustomerID = s.CustomerID,
               i.OrderDate = GETDATE(),
               i.PurchaseOrderNumber = s.PurchaseOrderNumber,
               i.TotalDue = s.TotalDue
WHEN NOT MATCHED THEN
    INSERT (SalesOrderID, CustomerID, OrderDate, PurchaseOrderNumber,
TotalDue)
    VALUES (s.SalesOrderID, s.CustomerID, s.OrderDate, s.PurchaseOrderNum-
ber, s.TotalDue);
```

Lab - Modify Data

Estimated time: 30 minutes

Depending on the lab hosting solution used for your class delivery, the lab instructions may display directly in the virtual lab environment. In addition, they are available **online in GitHub**[1]. Follow guidance from your instructor to access the lab environment.

Module-Review

Module Review

Use the questions below to check your knowledge of the topics covered in this module.

1 https://microsoftlearning.github.io/dp-080-Transact-SQL/

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Multiple choice

You want to insert data from the Store.Product table into an existing table named Sales.Offer. Which statement should you use?

- ☐ INSERT INTO Sales.Offer SELECT ProductID, Name, Price*0.9 FROM Store.Product;
- ☐ SELECT ProductID, Name, Price*0.9 FROM Store.Product INTO Sales.Offer;
- ☐ INSERT INTO Sales.Offer (ProductID, Name, Price*0.9) VALUES (Store.Product);

Multiple choice

You need to determine the most recently inserted IDENTITY column in the Sales.Invoice table. Which statement should you use?

- ☐ SELECT SCOPE_IDENTITY() FROM Sales.Invoice;
- ☐ SELECT IDENT_CURRENT('Sales.Invoice');
- ☐ SELECT NEXT VALUE FOR Sales.Invoice;

Multiple choice

You must increase the Price of all products in category 2 by 10%. Which statement should you use?

- ☐ UPDATE Store.Product SET Price = Price * 1.1, Category = 2;
- ☐ UPDATE Store.Product SET Price = Price * 1.1 WHERE Category = 2;
- ☐ SELECT Price * 1.1 INTO Store.Product FROM Store.Product WHERE Category = 2;

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED

Answers

Multiple choice

You want to insert data from the Store.Product table into an existing table named Sales.Offer. Which statement should you use?

■ INSERT INTO Sales.Offer SELECT ProductID, Name, Price*0.9 FROM Store.Product;

☐ SELECT ProductID, Name, Price*0.9 FROM Store.Product INTO Sales.Offer;

☐ INSERT INTO Sales.Offer (ProductID, Name, Price*0.9) VALUES (Store.Product);

Explanation
Use INSERT ... SELECT to insert the results of a query into an existing table

Multiple choice

You need to determine the most recently inserted IDENTITY column in the Sales.Invoice table. Which statement should you use?

☐ SELECT SCOPE_IDENTITY() FROM Sales.Invoice;

■ SELECT IDENT_CURRENT('Sales.Invoice');

☐ SELECT NEXT VALUE FOR Sales.Invoice;

Explanation
Use IDENT_CURRENT to find the current Identity value in a specific table.

Multiple choice

You must increase the Price of all products in category 2 by 10%. Which statement should you use?

☐ UPDATE Store.Product SET Price = Price * 1.1, Category = 2;

■ UPDATE Store.Product SET Price = Price * 1.1 WHERE Category = 2;

☐ SELECT Price * 1.1 INTO Store.Product FROM Store.Product WHERE Category = 2;

Explanation
Use UPDATE with a WHERE clause to update specific rows.

AUTHORIZED TRAINER USE ONLY. STUDENT USE PROHIBITED