TABLE OF CONTENTS

TABLE OF CONTENTS

Chapter 1: Overview of SSMS and Query Writing

Review datasets within SSMS 1-2
Using the SQL Editor 1-4
Creating SSMS script projects 1-9
Tips and tricks with SSMS 1-11
Adding comments to queries 1-15
Understanding batches and scripts 1-16
Importing/Exporting Data 1-19

Chapter 2: The SELECT Statement

The SELECT Statement 2-2
Execution Order of SELECT Statements 2-7
Expressions 2-9
Ordering Results 2-18
Filtering Rows 2-20
Comparison Operators 2-20
Logical Operators 2-24
Additional SELECT Options 2-35

Chapter 3: Built-in Functions Overview

How to find help on functions 3-2
Working with Functions 3-6
Mathematical Function Overview 3-6
String Function Overview 3-10
Date Time Function Overview 3-17
Nesting Functions 3-32
Understanding Data Type Conversion 3-34

Chapter 4: Handling NULL Data

NULL vs blank 4-2

= vs IS NULL 4-3
ISNULL function 4-5
COALESCE 4-5
Concatenating NULL data 4-8

Chapter 5: Aggregating and Grouping Data
Aggregate functions 5-2
GROUP BY 5-5
HAVING 5-8
HAVING vs WHERE 5-9
Overview ROLLUP and CUBE 5-11
OVER with Aggregates 5-14
OVER with Ranking Functions 5-17

Chapter 6: Joining Multiple Tables
JOINS 6-2
INNER JOIN 6-2
OUTER JOIN 6-6
CROSS JOIN 6-12
Joining Three or More Tables 6-12
Self-join 6-13
Alternate Syntax, Implicit Joins 6-16
Set operations 6-16
Viewing graphical execution plans 6-20

Chapter 7: Subqueries
Subqueries 7-2
Nested vs Correlated Subqueries 7-2
Subqueries in the SELECT Clause 7-4
Subqueries in the WHERE Clause 7-6
EXISTS 7-9
Subqueries in FROM Clause 7-9
Alternatives to Subqueries 7-13

 Copyright © 2018 Ann L. Weber

Chapter 8: Importing Data

Import/Export Wizard 8-2

Exporting Data with the Wizard 8-18

Understanding Data Types 8-19

Common Import Concerns 8-19

Quality checking imported/exported data 8-22

Chapter 9: Data Manipulation Language

Transaction Overview 9-2

Insert 9-5

INSERT SELECT vs SELECT INTO 9-7

Update 9-10

DELETE 9-14

Chapter 10: Data Definition Language

Creating Tables 10-2

ALTER TABLE 10-5

DROP TABLE 10-7

Creating indexes 10-8

DROP INDEX 10-12

When to use indexes 10-12

Using the Graphical Execution Plan and Missing Index Hints 10-12

Chapter 11: Working with Temporary Objects

Declaring variables 11-2

Importance of using correct data types 11-3

Table variables 11-4

Temporary Tables 11-6

Common Table Expressions (CTEs – If time permits) 11-7

Chapter 0 - Introduction

In this chapter:

Program Overview
Datasets Overview

Files needed:

- \Chapter 0\Diagrams

Program Overview

This two week training session focuses on SQL query writing, Tableau, and Python scripting, and will utilize one or more of the following three datasets. Additionally, as you work through these three platforms, you will solve many of the same questions and objectives using one or more of the tools listed above.

General Goals

Over the next two weeks, you will use three very different tools to answer many common data related questions. Part of the learning process will include understanding the power behind each of these tools and learning which tool is best for any given job. Sometimes, two tools may be equal to the task and the tool you choose will be a matter of preference or availability. Other times, you might be able to achieve your goal using one tool, but another tool would have been a much better choice. You may be able to use the back of a screwdriver to hammer in a nail, but it isn't the best tool for the job.

Some general learning goals across the classes include:

- Importing data provided as csv files or other formats
- Exporting results
- Learning how to interpret data
- Looking for outliers, trends, and patterns

 Copyright © 2018 Ann L. Weber

Data Manipulation Verbs

When working with data, no matter what the language, you are frequently accomplishing many of the same general tasks that can be summarized using a short list of "verbs". The following list encompasses most of the tasks you will be learning and performing over the next two weeks.

Data Manipulation Verb List:

1) Import file (not technically manipulation, but often step one)
2) Create new columns
   a) Create new column in dataset without changing the row count
3) Transform existing columns (by applying functions or operators)
4) Sort data
5) Select columns (subset data)
6) Filter rows (subset data)
7) Summarize
   a) Apply summary functions to one or more columns
8) Group by
9) Reshape
   a) Go from long to wide format
10) Merge two datasets
    a) Merge by a common key
11) Concatenate data
    a) Stack one dataset on top of another

Example Usage:

- Question: Who is the highest paid staff member for each job category?
- Answer:
  - Import staff table with salary information
  - Group by job category
  - Summarize salary by calculating max salary (Within job category group, because we already said group by)
  - Filter to where salary equals max(salary)
  - Select Staff Name, job category

Courseware Overview

Most chapters of the course materials include Try It exercises throughout the chapter. There will be a warning in the chapter introduction if the exercises are dependent on one another. Otherwise, each Try It exercise stands on its own. Each chapter is always independent of other chapters.

Scripts are provided for all of the inline samples within each chapter's content. Typically, there is one script per chapter and a comment preceding the sample will correlate with the section heading or purpose of the sample.

Most chapters will end with a lab made up of one or more exercises. These labs will give you the opportunity to work independently, practicing what you learned in that chapter and also incorporating topics from previous chapters. Each step in the lab should provide you with sufficient information to accomplish the task if you are comfortable with the content up to that point. If you find you need help, the top level directions are repeated using the same numbering scheme in the Lab Answer Key section that follows each exercise. The top level primarily tells you **"What"** you are to accomplish. In the lab answer key, the lower levels, represented with parentheses like (1), tell you "**How**" you will perform the goal in the top-level directions.

The final chapter of the course materials is one large lab where you will be given a set of goals to achieve. In this chapter, there will be less structure in the first section of the lab directions to allow you to explore and start to adjust to working without a book to guide you through the thinking process. Although there will not be step by step directions in the lab answer key, there will be a set of script files to demonstrate one way to achieve the stated goals. The book answers are rarely the only possible correct answers.

Class Files

The recommended path for the class files is **C:\ Classfiles\T-SQL\...** . Within this folder structure, the stated path in the book is usually a relative path starting below this T-SQL folder to avoid confusion if your class files were deployed to a different location such as a network drive. For example: \Chapter 0.

Under the T-SQL folder, the following structure is followed:

- \Chapter xx
    - \Inline Samples
    - \Try It Exercises
    - \Labs
- \Student Files

All files that are created during the class will be saved to the \Student Files folder. This folder is located at the same level as the Chapter *XX* folders, directly under the T-SQL folder. Although the course does not specify making sub-folders under Student Files, you can do so if you like.

Datasets Overview

Although these datasets have been simplified to improve the learning experience and focus on the tools and data processes being learned, the datasets come from three industry areas that you are likely to see during your career. At times, these datasets are purposefully designed poorly or include "bad data" to better illustrate certain lessons.

The class datasets were generated via scripts and contain no "real" data. Due to the nature of generating random data, you may find trends, etc. that are inconsistent with "real world" data.

 Copyright © 2018 Ann L. Weber

Retail Banking Sample Data

The retail banking dataset includes eight tables that track customers, their accounts, and banking transactions. An Entity/Relationship (ER) diagram representing this dataset can be found in Figure 1 and is located in the **RetailBankingSample ERD.png** file in the **\Chapter 0\Diagrams** folder. You can use this diagram through the class to determine what fields are available to provide the data requested in the instructor led practices and self-guided labs. Most of the inline chapter samples and Try It exercises will use this database.

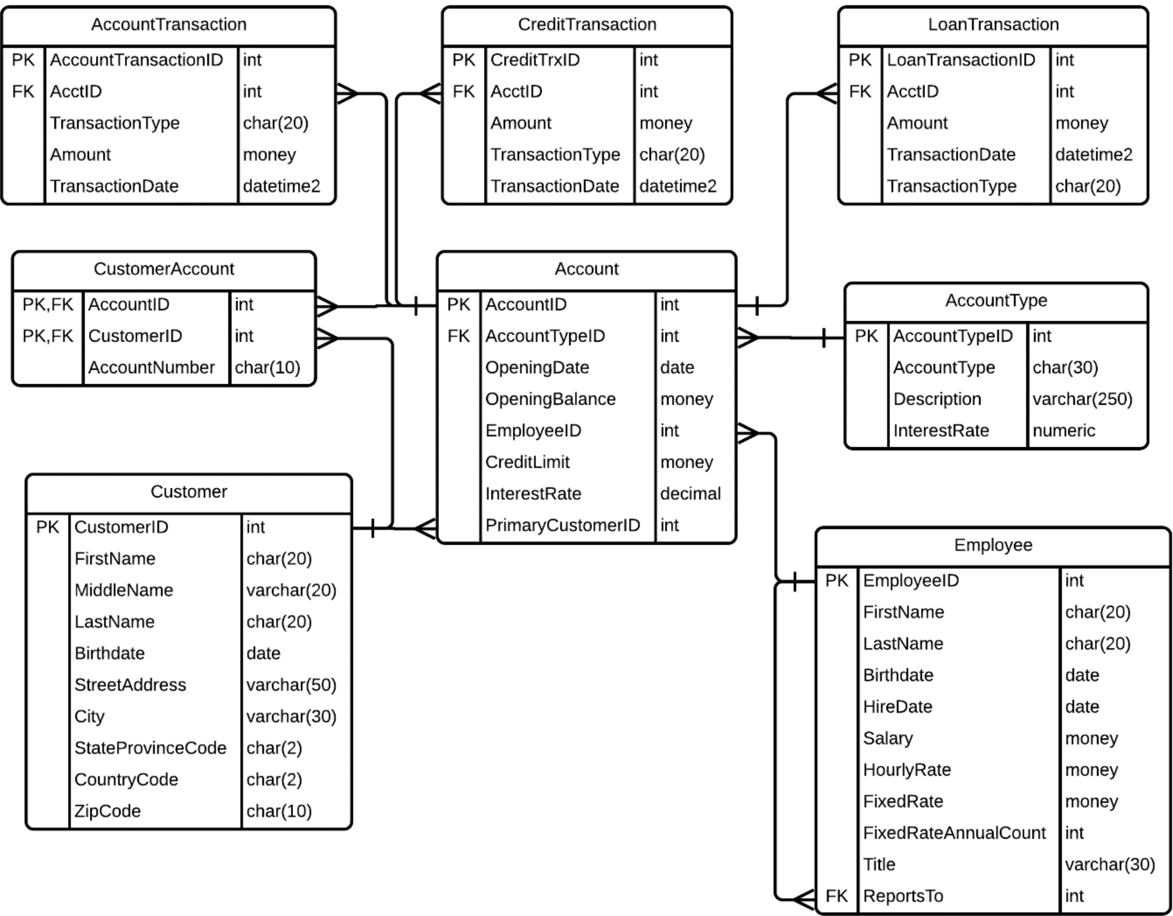


Figure 1: Retail Banking Entity/Relationship Diagram

Table Descriptions

Although the table names are fairly self-descriptive, the following list describes each table and any special features within the list.

- Customer: This table uses the CustomerID field to uniquely identify each bank customer. Additionally, the Customer table contains fields to hold the customer's name, birthdate, and address information.

- Account – The Account table uses and AccountID to uniquely identify each account. Addition fields describe the type of account and information about the account itself. Not every field relates to every account type, so some fields allow NULL values.
- AccountType - To maintain 3rd normal form, the AccountType table includes an AccountTypeID to uniquely identify each row, the actual account type and a description of the account type.
- CustomerAccount – The CustomerAccount table includes a composite key made up of the CustomerID and the AccountID. This table allows SQL to handle the situation where an account has more than one customer associated with it, while also allowing customers to be linked to multiple accounts. Because many-to-many relationships are not directly supported in SQL, this table facilitates that functionality. Transactions though are only linked to accounts, not individual customers. You cannot track who spent what in this particular database.
- The **Employee** table includes an EmployeeID to uniquely identify each employee along with fields that contain additional information about the employee such as their name, hire date, birthdate, payment information, and more.
- AccountTransaction, LoanTransaction, and CreditTransaction – Due to the differences in how credit card, loan, and standard banking accounts track information and apply negative and positive values, these three tables were created rather than one single table that holds all transactions.
  AccountTransaction holds information about checking and savings accounts, while CreditTransaction holds credit and debit card transactions and LoanTransaction holds loan payments and interest accrued.
  Additionally, the tables are denormalized in that they have the transaction types included in this table rather stored in a separate table. This is not a best practice, but rather a representation of things you may see in the databases you work with.
  Each transaction table includes a unique primary key field as well as fields to describe the transaction itself and includes the amount, date, and type of the transaction. The AcctID field relates back to the Account table. Through this relationship and other relationships in the database you will be able to determine the customer's name and account information as well.

Sample Questions for RetailBankingSample Database

- What customers had transactions in 2007 but do not have any transactions this year?
- What percent of customers have both savings and checking accounts?
- How many accounts have two or more customers linked to them?

HealthCare Sample Data

The HealthCare sample dataset is made up of seven tables that track staff, patients, diagnoses, and more. Most of the end of chapter labs will reference this database.

 Copyright © 2018 Ann L. Weber

An ER diagram representing this dataset can be found in Figure 2 and located in the **HeathCareSample.png** file in the **Chapter 0\Diagrams** folder.

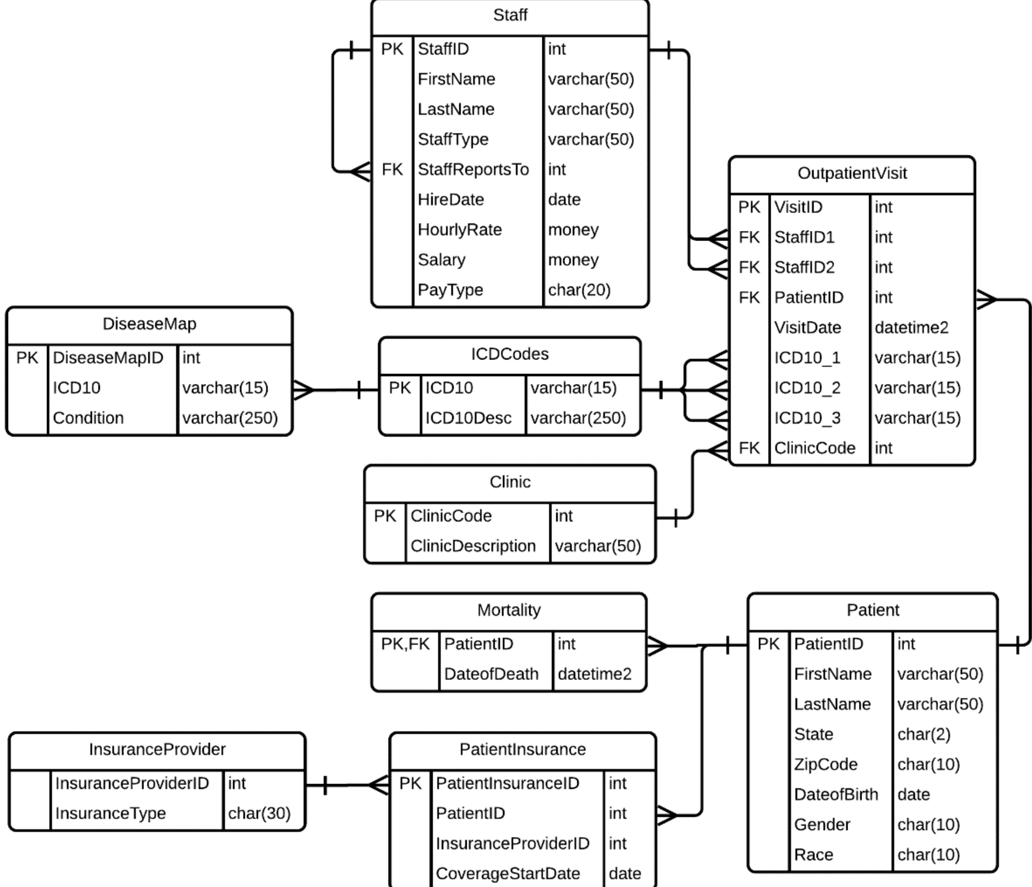


Figure 2: Healthcare Entity/Relationship Diagram

The following list describes the tables in the Healthcare sample dataset.

- The Patient table includes the patient's name, insurance, and demographic information. The PatientID is a unique key automatically generated and is used to relate the patients to the Mortality and OutpatientVisit tables.
- The Mortality table includes only the PatientID and DateofDeath of patients who have passed away. The relationship between these two tables is a one to one relationship, but to improve lookups and reporting, the DateofDeath was moved to a separate table.
- The Staff table tracks the name, staff type, and supervisor for any staff involved with patient visits.
- The Clinic table provides a description of the type of clinic associated with the ClinicCode on the OutpatientVisit table.

- The OutpatientVisit table links a patient with a clinic, one or more staff, and up to three diagnoses on a particular date. The ICD10 codes are standard codes that map to certain diseases. At each visit, the patient can have up to three diagnosis. The ICD10_1, ICD10_2, ICD10_3 columns are populated in order based on the number diagnoses.
- The ICDCodes table lists the ICD10 codes and descriptions for each diagnoses.
- The DiseaseMap table includes a DiseaseMapID which is an auto-generated number to provide a primary key for the table. This table maps each diagnosis to one or more diseases that it may be associated with. Each diagnosis may be listed multiple times in this table if it is part of more than one disease.
- The PatientInsurance table links patients to insurance providers along with a coverage start date.

Sample Questions for HealthCareSample Database

- How many patients have an IDC10 code in one of their visits that related to a Stroke?
- What percent of all patients have a date of death in the years 2016-2018?
- How many patients were seen in the past but not in 2017 or 2018?

Phishing Detection Data

The phishing detection dataset is designed to track auditing campaigns where companies try to determine how susceptible they are to phishing attacks. The phishing detection dataset is made up of the three following tables:

1) User – information pertaining to users that received or reported spam/phishing emails.
2) Campaign – information about the type of phishing email that was sent and the date on which it was sent.
3) Lookup – information connecting users to specific emails that were opened or reported.

An ER diagram representing this dataset can be found in Figure 3 and located in the **PhishingSample ERD.png** file in the **Chapter 0\Diagrams** folder.

 Copyright © 2018 Ann L. Weber

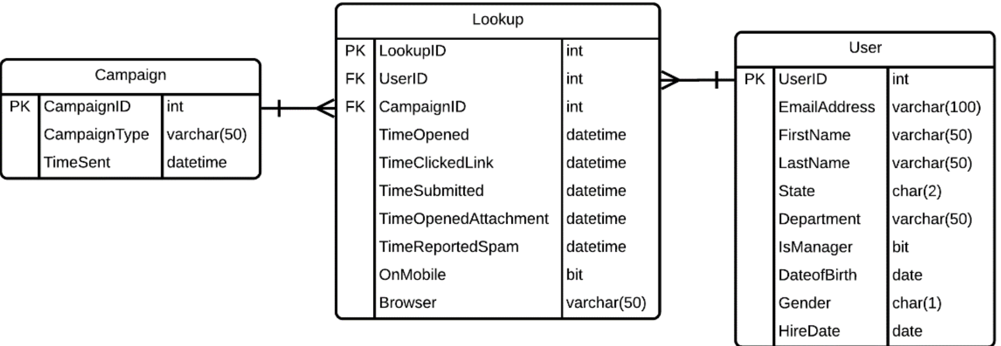


Figure 3: Phishing Detection Entity/Relationship Diagram

Sample Questions for Phishing Sample Database

- What percent of the users reported the email?
- What are the longest and shortest lags, measured in minutes, between the time the email was opened and when it was reported?
- Was there anyone who reported the email without opening it? If so, what are their names, email addresses, and departments?

Chapter 1 - Overview of SSMS and Query Writing

In this chapter:

Review datasets within SSMS
Using the SQL Editor
Try It 2 – The Query Editor
Creating SSMS script projects
Try It 3 – Creating and Using a Script Project
Tips and tricks with SSMS
Adding comments to queries
Understanding batches and scripts
Importing/Exporting Data

Files needed:

- \Chapter 01 SSMS\Try It Exercises
- \Chapter 01 SSMS\Inline Samples
- \Student Files

	There is no lab in this chapter. Some of the Try It Exercises in this chapter build on one another, but are independent of other chapters. Any starter or answer files for the Try It exercises can be found in the \Try It Exercises folder for each chapter. All code samples included within the chapter text are located in the \Inline Samples folder for each chapter.

 Copyright © 2018 Ann L. Weber

Although there are many programs available from Microsoft and other companies to write Microsoft Transact SQL (T-SQL) queries, this class focuses on SQL Server Management Server, known simply as SSMS.

Review datasets within SSMS

Each time you open SSMS, you must decide the SQL Server instance type, the server name, and security connection information that will be used to connect the Object Explorer in SSMS to the correct instance. On any given physical computer or virtual machine, you can install multiple instances of SQL Server. The instance type we will be working with in this class is the SQL Relational Database Engine. Typically, in the classroom, you will be working on a single instance of this database instance. This is what we refer to as a SQL Server when speaking generically.

The SSMS icon looks like a yellow cylinder representing a database connected by dotted lines to a wrench and hammer. Depending on your operating system and short-cuts, the method of launching the application may vary.

Object Explorer

As soon as SSMS launches, you are prompted to enter connection information to an instance of SQL Server. Once you are connected, SSMS will open the Object Explorer. The Object Explorer offers multiple ways to view your database schemas and data. Much like File Explorer in Windows, Object Explorer organizes your SQL Server instances into a tree structure. The icons represent the types of objects available. You can browse down through the levels (referred to frequently as “folders” regardless of the icon type) to see databases, tables, columns, and much more. You can connect to one or more SQL Server instances located either on your local machine or on a remote host.

Database Diagrams

Database diagrams offer anyone with sufficient permissions a way to view and modify the database schema. Some caution must be exercised here. For example, when you right-click on a table, there are two options that can sound similar if not fully understood. The first option, **Delete Tables from Database,** will completely drop the table and delete all data in the table. The second option, Remove from Diagram, only changes the visibility of the table in the diagram. It does not affect the underlying table structure or data.

In order to set up the database diagram feature in a new database, a user must be a member of the db_owner role. Once the feature is configured, anyone can create a database diagram, but only users with the appropriate permissions can make changes through the database diagram tool.

Viewing Data

In addition to writing queries, you can use right-click options within the Object Explorer to view and modify data within your tables.

Try It 1 – Opening SSMS and Connecting to Object Explorer

<table>
<tr>
<td>
</td>
<td>Each topic will include a few brief steps to try along the way. Most of these practices will be guided by your instructor. Unless otherwise noted, each Try It practice in this chapter will build on the previous Try It practices and prerequisite steps will <u>not</u> be called out or repeated.

Please let your instructor know if you fall behind or need help.</td>
</tr>
</table>

1. Launch SSMS.
   a. For Windows 10, click on the Window icon in the bottom left corner of your screen, and then start typing **SSMS**. SQL Server Management studio should appear under the Best Match heading.
2. Right-click **SQL Server Management Studio**, and then select either **Pin to start** to place a shortcut on the large Start tiles or **More | Pin to taskbar** to place the shortcut on the taskbar at the bottom of your screen.
3. Click your new shortcut to launch SSMS.
4. In the Connect to Server dialog box, verify the following options, and then click **Connect**.
   Note: If you are not running the database engine on your local computer, enter the appropriate server name and authentication information for your environment.

Field	Value
Server type	Database Engine
Server name	(local)
Authentication	Windows Authentication

5. Object Explorer should automatically open on the left side of SSMS. If it does not, click **View | Object Explorer.**
6. Click the + (plus sign) next to the Databases folder under your local SQL Server instance to view the databases on your computer.
7. Notice the three databases introduced in the previous chapter.
8. Click the plus sign next to the **RetailBanking** database to expand the subfolders.
9. Expand the **Database Diagrams** folder, and then double-click the **RetailBankingFull** diagram.

<table>
<tr>
<td>
</td>
<td>These diagrams are <u>NOT</u> read-only. If you have the privileges, you can accidentally delete an entire table and all of its data. Please be very careful when using these diagrams.</td>
</tr>
</table>

 Copyright © 2018 Ann L. Weber

10. Review the tables, columns, and relationships between the tables. Notice that the three transaction tables do not have relationships directly to the Customer or CustomerAccount tables. Transactions are only related to the primary customer, which is found through the Account table.
11. Close the database diagram.
12. In Object Explorer, expand **Databases | RetailBankingSample | Tables**.
13. Right-click the **Account** table, and then click **Select Top 1000 Rows**. Review the data returned. Notice the PrimaryCustomerID column.
14. In Object Explorer, right-click the Customer table, and then click Edit Top 200 Rows. This window will allow you to make changes to the first 200 rows in the table as well as enter new rows.
15. Repeat the process above to review the PhishingSample and HealthCareSample databases. The HealthCareSample database will be used in most of the labs in this book. The PhishingSample database is used from time to time when the data provides a more relevant sample for a topic.

Using the SQL Editor

The SQL Editor opens when you click the New Query icon or use one of the many other methods to create a new query, such as right-clicking a database, and then selecting New Query. Most of the common options for the query editor can be found in each of the three locations:

- The Query menu shown in Figure 4.
- The SQL Editor toolbar shown in Figure 5.
  - If this toolbar is closed, click **View | Toolbars | SQL Editor** to re-enable it.
- The right-click menu, which is accessed by right-clicking the area inside of the query editor as shown in Figure 6.

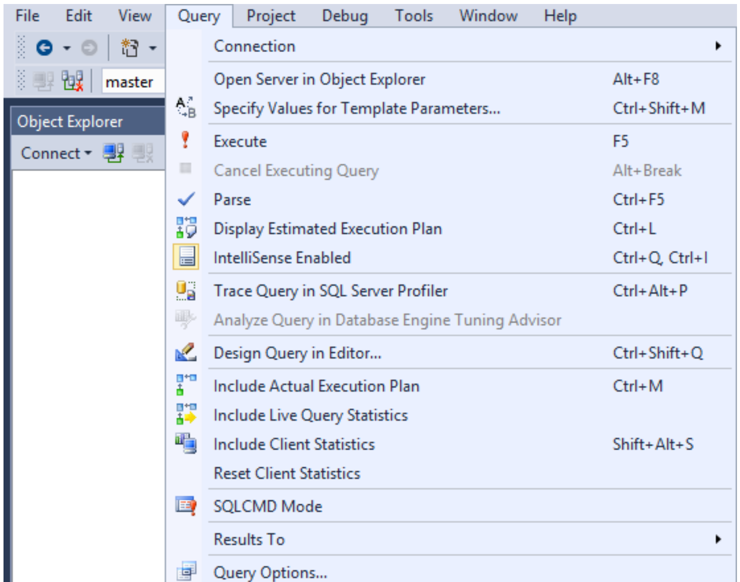


Figure 4: The Query Menu

Figure 5: The SQL Editor Toolbar

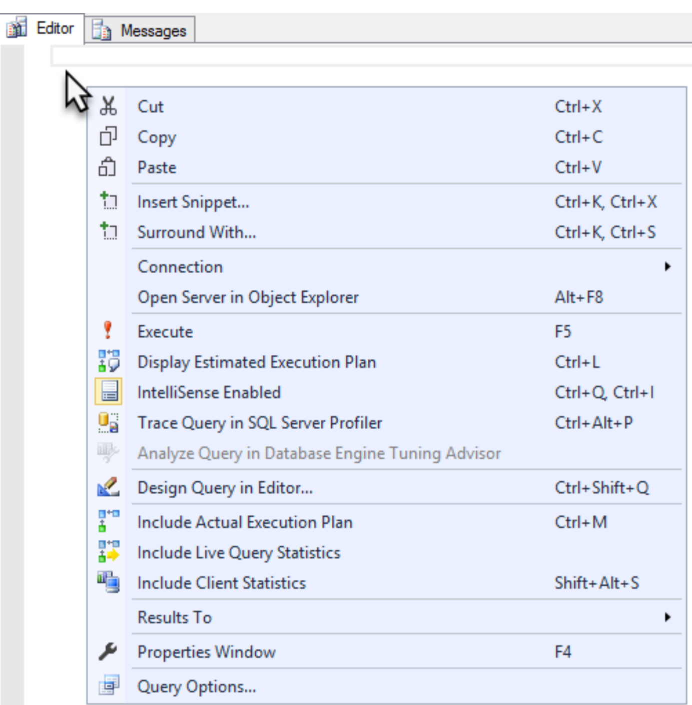


Figure 6: Right-Click Menu

 Copyright © 2018 Ann L. Weber

Query Tab Right-Click Menu

Additional options are available when you right-click the tab at the top of the query editor. If you want to close a single tab, click the X on the right side of the tab. If you want to close all tabs other than the one you are currently working on, right-click the tab you are working on, and then click **Close All But This**. You can also quickly open the folder containing a saved query or copy the full path by using the right-click menu.

Closing, Hiding, and Floating Windows

If you are working on multiple queries simultaneously and have multiple monitors or one very large monitor, you can Ctrl + double-click the tab to float the SQL Query Editor and move it independently from the rest of SSMS. Once it is floating, Ctrl + double-click will dock it back as a tab.

	Depending on your version of SSMS, you may not need to hold the Ctrl key for either the docking or the undocking process.

To give yourself more room in the query window, you can use the pushpins to auto-hide the Object Explorer and Solution Explorer. When the push pin is vertical, the window is permanently in place. When the push pin is horizontal, the window is represented by the title bar and can be opened when you click on it. A window with auto-hide will likely cover your query when active, but will go away once you click in the SQL Query Editor.

Any window can be closed with the X in the upper-right corner and then opened again from the View menu.

Saving Queries

If your cursor is in the query portion of the window when you click the Save icon, the query will be saved. If the query has not yet been saved, the Save File As dialog will open and the Save As type drop-down list will be (*.sql).

The default location for query files is the \Documents\SQL Server Management Studio folder. For class, you will be saving your scripts to **\Classfiles\T-SQL\Student Files**. This folder is in the same parent folder as the Chapter xx folders that hold the files for each chapter. Throughout the rest of the class, these folders will be abbreviated as **\Student Files** or **\Chapter xx\subfolder name**.

Saving Results

Your query results can be saved or exported in a number of ways. The "save results" dialog box can be activated by:

- **File | Save Results As**
- Right click in the results area, then click **Save Results As**

This dialog box varies based on whether the query was run in grid mode or text mode.

When the query is run in Results to Grid mode, the Save Grid Results dialog box opens and the Save as type option defaults to CSV (Comma delimited) (*.csv). You can also select the type option Text (Tab delimited) (*.txt). If you select All files (*.*) option and create your own extension, the file will be tab delimited. When you save results from the result window in the Results to Grid mode, the column headings are not included by default. You can change this behavior through **Tools** | **Options** | **Query Results** | **SQL Server** | **Results to Grid** | **Include column headers when copying or saving the results**. Although this option works for saving files, it does not work for copying from the results window. For copying, you must use one of the methods discussed below.

When the query is run in Results to Text mode, the Save Results dialog box opens and the Save As type option defaults to Report files (*.rpt). The only other option is All files (*.*). The file does not include delimiters, but each column is a fixed width based on the data type of the column.

If you run a query with the **Results to File** option selected, when you execute the query the Save Results dialog opens with the same options as with the Results to Text mode.

In addition to the numerous save options, you can use the Import/Export Wizard to export the results of a query to any number of destinations including flat files, databases, and Excel files. You will see a demonstration of the Import/Export Wizard later in this chapter and then you will have a chance to work with the wizard in **Chapter 8 Importing Data**.

One last option for exporting results is the good old copy and paste method. By default in the Results to Grid mode, when you copy results from the result window, the column headings are not included. The heading can be included by using any of the following methods:

- Right-click in the Results window and select **Copy with Headers**.
- **Edit** menu | **Copy with Headers**
- Press **Ctrl + Shift + C**

Using Exported Data

Once you have saved or exported your result set, the method you use to open that result set may change your data. For example, if you have a zipcode field that uses a character data type in the database to preserve leading zeros, once you export the data to a csv and then open the same file with Excel will cause you to lose the leading zeros. Alternatively, using Notepad or a similar program will preserve the leading zeros.

There are work-arounds to this behavior. Little tricks such as opening a new Excel document and formatting all the columns as text before pasting your results or adding a text qualifier such as double quotes to the data with the Import/Export Wizard and then specifying the data type of each column within the Excel import wizard will allow you to retain the leading zeros on the zipcodes and other similar concerns.

 Copyright © 2018 Ann L. Weber

Try It 2 – The Query Editor

In this exercise you will practice creating a new query, saving that query, and then saving the result set by using a variety of methods.

1. If necessary, open SSMS and connect to your class instance of SQL Server with the appropriate credentials. Typically this will be the Database Engine, (local) or a period "." for the server name and Windows Authentication.
2. Click the New Query button to open a new Query Editor window.
3. Click **File | Save** or the Save icon.
4. Save the file as **Ch1TryIt2.sql** in the \Student Files folder. If the default setup directions were followed, this folder will be located at C:\Classfiles\T-SQL\Student Files.
5. Either select RetailBankingSample from the database drop-down list as shown in Figure 7, or type the following command in the query editor.

```
USE RetailBankingSample;
```

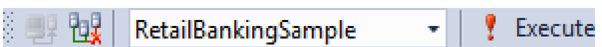


Figure 7: Database Drop-down List

6. Type the following command, and then click the Execute button or press F5 to run the query.

```
SELECT *
FROM Account;
```

7. Below the existing query, type the following query. Highlight just the new query, and then click the Execute button or press F5. This will run just the highlighted portion of the code.

```
SELECT zipcode
FROM Customer;
```

8. With the cursor in the top section with the code, click the Save icon. Your query changes will be saved to the file you created in step 4.
9. Right-click the results area, and then click **Save Results As**.
10. Save the file as a csv file with the name **Ch1Try2** in the\Student Files folder.
11. Minimize SSMS and open **File Explorer** (may also be referred to as My Documents, My Computer, Windows Explorer).
12. Browse to the \Student Files folder.
13. If you are running Windows 8 or later, click the **Pin to Quick access** icon on the Home tab of File Explorer. In Windows 7 Windows Explorer, you can right-

Copyright © 2018 Ann L. Weber

click the Favorites item at the top of the browse list and select Add current location to Favorites.

14. Double-click the **Ch1Try2.csv** file. This should launch Excel and open the file. Notice that some of the zip codes are only 4 digits. Close Excel **<u>without</u>** saving.
15. Right-click the **Ch1Try2.csv** file, and then click **Open with | Notepad**. Notice that in notepad, the leading 0s are still in the results. They just disappear once opened in Excel.
16. Close Notepad.
17. Return to SSMS.
18. Click in the result set, click CTRL+A to select the entire result set, then click Ctrl + Shift + C to copy the contents with the headers.
    Note: You can also use the right-click menu to select everything and then copy with headers.
19. Open Excel and a new workbook.
20. Paste the data into the first column. Notice that the column headers ARE there, but the leading 0s are NOT there.
21. Undo the paste.
22. In Excel, select column A.
23. On the Home Tab in the Number section, change the format drop-down window to Text.
24. Paste the results into A1. The leading 0s should be maintained.
25. Close Excel and File Explorer.
26. Close your current query window but leave SSMS open for the next Try It.

Creating SSMS script projects

When you are working on a project with a large number of script files, use SSMS Script Projects to organize and maintain your scripts. Each project / solution can contain one or more connections. Each connection defines a SQL Server instance plus authentication information. For each script within the project, set the database context for that script either by using the drop-down list, or preferably, by including a USE *database* command in the script.

Additionally, any script that you create in the project will be available both as an individual file and associated with the project. The scripts are listed in alphabetical order.

	When I have a lot of scripts in a project that need to be run in a particular order, I will usually number them so that they are sorted in the order they are needed. When things change, as they always do, I end up with steps that start 3a, 3b, 3c, etc. So far, I haven't run out of letters for new steps that need to go between existing steps.

 Copyright © 2018 Ann L. Weber

Try It 3 – Creating and Using a Script Project

In this exercise, you will practice creating a Script Project in SSMS.

	Most of the labs will use script projects, while the Chapter Try It exercises will typically use stand-alone or single script files. This will give you practice with both methods. Unless your company has a preferred method, you can choose the method that best suits the needs for your project.

1. If necessary, open SQL Server Management Studio (SSMS).
2. Click **File | New | Project**
3. In the New Project dialog box, verify that **SQL Server Scripts** is selected in the templates area in the center.
4. Change or verify the following settings, and then click **OK**.
    a. Set the Name to **Ch1Ti3.**
    b. Change the location to the **\Student Files** folder.
    c. Verify that Create new solution is selected next to Solution.
    d. Verify that Create directory for solution is selected.
5. If Solution Explorer is not visible, click **View | Solution Explorer**.
6. In Solution Explorer, right-click the Connections folder, and then click **New Connection**.
7. Verify the Server name and Authentication information and then click OK.
8. Right-click the new connection that you just created and click **New Query**.
9. Under the Queries folder in Solution Explorer, right-click SQLQuery1.sql and click Rename.
10. Change the name to **SSMS.sql**.
11. Click the **Save All** icon, or click **File | Save All**.
12. Leave the solution open for the next Try It.

Using IntelliSense to your advantage

Within the SSMS Query Editor, when you start to type the name of an object within a database, the IntelliSense will pop-up a list of names that fit the pattern. Starting with SQL 2014, the algorithm uses a "contains" match rather than a "starts with" match making it even more beneficial.

A "trick" that you can use to help IntelliSense find what you are looking for is to perform the following steps:

1. Make sure that the current database is set with either a USE database command or by selecting it from the drop-down menu.
2. Type the word SELECT and then move down a line and type your complete FROM clause including an alias for the table name.
3. Return to your SELECT clause and type the alias followed by a period (.). The popup list will now be populated only with columns from that table or view.

4. If the column you want is selected, the Tab key will fill in the rest of that value. If it is not selected, keep typing or use your arrow keys to move to the correct name, and then press Tab. You can also use the mouse to select the correct item, but moving your hands off of the keyboard to the mouse will slow down your work significantly.

This course will typically use table aliases and specify the table alias in the SELECT clause, even though it is only required for a very small portion of the queries in this class. This allows for the use of this IntelliSense trick, and it improves readability of the code.

If the IntelliSense window opens but does not have what you want, you can use the escape (Esc) key to close the menu without changing what you typed. If you don't like IntelliSense, you can disable IntelliSense by changing the **Query** | **IntelliSense Enabled** toggle option.

Tips and tricks with SSMS

Although everyone has their own favorite settings and configurations in SSMS, this section will review how to set some of the most recommended options along with the author's favorites.

Refresh Local Cache

If you are using the IntelliSense feature, you may get frustrated by the red squiggly lines under newly created objects in the database. This is caused by the IntelliSense cache not being updated when the changes occur. If the IntelliSense pop-up window is already active, you must click outside the window to close it before updating the cache. If the pop-up window is open, the list will not be updated.

Here are two ways to update the IntelliSense cache: 1) type Ctrl + Shift + R, or 2) With the query window active, select Edit | IntelliSense | Refresh Local Cache.

Cycle Clipboard Ring

A great time-saving feature is the Cycle Clipboard Ring option available through the Edit menu or when you press Ctrl + Shift + V. Cycling the clipboard allows you to "walk" through the twenty most recent clipboard actions. (Ctrl + C or Ctrl + V)

Favorite Shortcuts

Ctrl + R	Open and close results pane
F6	Move cursor between all relevant areas of the current query tab, such as query, Results, Messages, and Execution Plan.
Ctrl + M	Include Actual Execution Plan
Ctrl + L	Get Estimated Execution Plan
Ctrl + Tab	Move forward through open query tabs
Ctrl + Shift + Tab	Move backwards through open query tabs

 Copyright © 2018 Ann L. Weber

Shift + Delete	Delete the entire current row
Ctrl + Z	Undo
Ctrl + Y	Redo
Ctrl + O	Launch Open File dialog box
Ctrl + Shift + O	Launch Open Project dialog box
Ctrl + N	Launch "new file" which typically is interpreted as new query editor tab.
Ctrl + Shift + N	Launch New Project dialog box
Ctrl + S	Save active item
Ctrl + Shift + S	Save all open items
Ctrl + Shift + Space	Display parameter hints

In addition to the shortcuts listed here, you can download a cheat sheet with popular SSMS, SQL Operations Studio (a new tool in preview at the time this course was written that makes for a better T-SQL coding experience), and other related shortcuts at: https://am2.co/2018/02/updated-cheat-sheet/. A full list of shortcuts can be found at https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-keyboard-shortcuts?view=sql-server-2017 and information on customizing menus and shortcut keys can be found at https://docs.microsoft.com/en-us/sql/ssms/customize-menus-and-shortcut-keys?view=sql-server-2017.

SSMS Configuration

Once you have your SSMS configured as you like it, you can export your current settings to a vssettings XML file and then import them to another computer. You can use your saved SSMS configurations to replicate your settings on additional computers and to place the settings in a shared location so that multiple users can import these same settings.

To either import or export settings, you will use the **Tools | Import and Export Settings** option in the menus.

Many of these options are configured by using the **Tools | Options** dialog box.

The list below includes some frequently configured options sorted by where in the menu structure the settings reside.

Tools | Options by section

Environment | General

- Change the number of items shown in recently used list

Environment | Fonts and Colors

- Change the display fonts for any part of the user interface

Text Editor | All Languages

- Choose whether or not to display line numbers

Text Editor |Transact-SQL | IntelliSense

- Enable IntelliSense and configur IntelliSense options

Query Execution | Advanced

- Specify persistent advanced execution setting such as NOCOUNT, STATISTICS TIME, STATISTICS IO, etc.

Query Results | SQL Server | General

- Change default save location

Query Results | SQL Server | Results to Grid

- Display results in a separate tab
- Include the query in the result set
- Include column headers when copying or saving results

Designers | Table and Database Designers

- Manage options when creating or altering objects using the graphical interface

	Use extreme caution if you turn off the "Prevent saving changes that require table re-creation" option. Although the server attempts to avoid data loss by moving the data into a temp table, this feature will allow you to inadvertently drop tables and delete data without warning.

Try It 4 – Working in SSMS

In this exercise you will explore some of the tips and tricks for using SSMS.

1. Verify that **Ch3Ti3.ssmssln** is open in Solution Explorer in SSMS. If you can't see Solution Explorer, click **View | Solution Explorer**.
   Note: If you did not complete Try It 3, click **File |Open | Project\Solution**, browse to and select **Chapter 01 SSMS\Try It Exercises\Try It 3 - Script Project TI 4 Starter\ Ch1Ti3\Ch1Ti3.ssmssln**, and then click **Open**.
2. Verify that the **SSMS.sql** query editor window is open. If the query editor is not open, double-click SSMS.sql under the Queries folder in Solution Explorer.
3. We will use line numbers to locate code in a sample exercise later in this chapter, but line numbers are not turned on by default. To enable line numbers, click and expand **Tools | Options | Text Editor | All Languages**.
4. If necessary, select the General page under All Languages.
5. Click Line numbers to enable line numbers for all languages.
   Note: An empty box means the feature is not enabled, a check mark means it is enabled for all languages, and a solid black square means that the feature is enabled for some but not all of the languages.
6. In Tools | Options, change to **Query Results | SQL Server | Results to Grid**.

 Copyright © 2018 Ann L. Weber

7. On the Results to Grid page, select **Include column headers when copying or saving the results**. If desired, select **Include the query in the result set**. Remember, this includes what you just executed in the Messages tab rather than the Results tab.
8. Explore some of the other options available under Tools | Options.
9. In the **SSMS.sql** query editor, type and then execute the following query. Remember to execute a query you can press **F5** or click the Execute icon.

```
USE RetailBankingSample;

SELECT * FROM Customer;
```

10. Right-click the result set and click Save Results As, browse to the **\Student Files** folder, and name the file **TryIt4Results.csv**.
11. Open the csv file and verify that the headers were created.
12. Click the Save All icon, and then click **File** | **Close Solution**. Leave SSMS open for the following Try It.

Adding comments to queries

Adding comments to scripts is not just about leaving information for someone else who may be maintaining or sharing your scripts. This is important, but comments also help you remember what you were doing when you come back after a break or after solving a separate problem. Understanding how to use comments can make writing complex queries and troubleshooting code easier.

Block Comments

Block comments mark out everything between the start indicator and the end indicator. Block comments are handy when you have a large section of non-code comments such as your name, editing date, purpose for the script etc. Block comments are also handy if you want to keep a large section of code in a script, but don't want to accidentally execute that portion of the script without highlighting it.

Although no special characters are required at the start of each new line, many programmers add one or more asterisks at the start of each new line to visually set the comments apart in some way other than by color.

Syntax

```
/*

*/
```

Sample

```
/* This is a sample of block comments.
Authors Name:
Date created:
Reason for script:
*/
```

Inline Comments

Inline comments start where a double-dash is typed and continues until a new line is started. Many of the supporting scripts for this class use inline comments.

You can use inline comments for the purposes above, but if using inline comments to comment multiple lines of code, all comments must be removed to execute the statement. You can easily add and remove comments with the icons on the SQL Editor toolbar.

Syntax

```
-- Sample Text
```

Sample

```
--Inline samples are used in the Inline Sample scripts
--Comment only lasts until a new line is started.
```

Understanding batches and scripts

Although some documentation makes it sound complicated, a script is nothing more than a set of one or more SQL commands. Typically, they will be saved as a .sql file. These files are normal text files, but the .sql extension associates the file type with SSMS.

A batch, on the other hand, affects how a query is executed. The word GO is called the batch directive. GO is not an SQL command and should NOT be followed by a semi-colon (;).

The following simplified list displays the step by step process that occurs every time you execute a query. The names of these phases vary depending on the resources and materials you are using.

- Parse – interprets the query and flags any syntax errors
- Resolve – Makes sure all names of objects exist in the current database context
- Optimize – locates a trivial or "good enough" execution plan
- Compile/Run

When a set of commands exist together in a single batch, all commands are parsed at the same time. A syntax error in any one command will cause the entire batch to fail and would never reach the resolve phase of query execution.

The behavior within a batch concerning logical and data errors (errors that SQL server will not encounter until running the command) will vary. Here are a few of the behaviors you could see:

- For a column name resolution error, no matter the column location within the script, the entire batch stops and no more commands are executed
- For a table name resolution error, the batch will begin to execute and you will receive result sets for all queries until an error is encountered. The batch will then stop executing immediately and will not attempt to run any queries after the error.

 Copyright © 2018 Ann L. Weber

- Logic/data errors will stop the execution at the point of the error. Because the query with the error starts to run, you may get column headings or other data. However, the query stops running as soon as an error is encountered. After the error, no more data is returned and no more commands are executed.

Certain SQL Statements such as CREATE VIEW and CREATE Procedure must be in a batch by themselves. You will frequently see these statements surrounded by GO keywords so that they can be executed independently or as part of a larger script.

```
GO
CREATE VIEW Myview
AS
SELECT 1 AS Col1, 2 AS Col2
;
GO
```

Try It 5 – Understanding Batch Directives

In this exercise you will see how adding a batch directive (GO) to a set of commands changes how resolving and syntax errors are handled. This exercise will also help you understand some of the different types of error messages you will see while running SQL scripts and commands.

1. In SSMS, click the Open File icon, or click **File | Open | File**.
2. Browse to **\Chapter 01 SSMS\Try It Exercises**, and open **Try It 5 – Batch Directives Starter.sql**.
3. Click **File | Save Try It 5 – Batch Directives Starter.sql As**, and then browse to **the \Student Files** folder. Type **Ch1Ti5.sql** in the File name, and then click **Save**.
4. Because the word GO has been left out of this script, the entire script performs as a single batch.
5. Execute the entire script. Review both the Results tab and the Messages tab.
6. Because of the data returned on the results tab, you can tell that the USE statement and first SELECT statement on lines 3 and 4 succeeded. The server then started the results section for lines 6 and 7, but ran into a conversion error in the very first record as noted on the Messages tab because the FirstName string cannot be converted into an integer.
7. Remove the comment marks from lines 12 and 13. You can do this by deleting them manually, highlighting the rows and either selecting the Uncomment the selected lines button, or holding the Ctrl key down then pressing K then U one after the other.
8. Verify that nothing is highlighted, and then execute the entire script. Notice that even though it is the very last query in the script, the invalid column name on line 12 causes the entire script to fail.

9. Comment out the SELECT CONVERT … statement on lines 6 and 7 by using a block comment as shown below:

```
/*
SELECT CONVERT(int, C.FirstName)
FROM Customer AS C;
*/
```

10. Modify the column name in step 12 to C.CustomerID as shown below.

```
SELECT C.CustomerID
From Customer AS C;
```

11. Verify that nothing is highlighted and execute the script.
12. Review the Results and Messages tab, noting that the first query did execute and returned results. However, everything following the query with the table resolution error did not execute.
13. Change the SELECT statement on line 12 so that the word FROM is spelled wrong. Execute the entire script. Notice that nothing runs because there is a syntax error somewhere in the script.
14. Add the word GO on line 11. Execute the query. Notice that the syntax error on line 13 no longer affects the previous queries and the resolution error on line 10 does not affect the final query. This is because there are now two batches and each batch runs independently from the others.
15. Save your script file and close the query window. Leave SSMS open for the next chapter.

Importing/Exporting Data

While analyzing data, you will frequently need to import and export data as part of the process. The SQL Server Import/Export wizard can ease this process.

Instructor Demonstration

Due to time constraints for the class, your instructor will follow steps similar to the ones below to demonstrate importing and exporting data by using the Import/Export wizard, which is launched from SQL Server Management Studio. In **Chapter 8 Importing Data**, you will have the opportunity to perform similar steps using this wizard. A full discussion on the Wizard’s functionality and SQL Server Integration Services (SSIS) is beyond the scope of this class.

1. Use SQL Server Management Studio to launch the Import Wizard from the RetailBankingSample database.

 Copyright © 2018 Ann L. Weber

   a. In Object Explorer, right-click the **RetailBankingSample**, and then click **Tasks | Export Data**.
2. Use the Wizard to retrieve data from the RetailBankingSample database by using the query found in **\Chapter 01 SSMS\Inline Samples\WizardExportDemo.sql**. Place the data in a comma separated flat file named EmployeeAccountsOpened.csv.
   a. On the Choose a Data Source page, select **SQL Server Native Client 11.0** in the Data Source drop-down and verify that the Database is set to RetailBankingSample before clicking **Next**.
   b. On the Choose a Destination page, select Flat File Destination, for the File name, click Browse and browse to the **\Student Files** folder and create a file named **EmployeeAccountsOpened.csv** and click **Open**. Click **Next** on the Choose a Destination folder.
   c. On the Specify Table Copy or Query, click **Write a query to specify data to transfer**, and then click **Next**.
   d. Either click the Brose button and locate the **\Chapter 01 SSMS\Inline Samples\WizardExportDemo.sql** file to import it, or open the query in either SSMS or notepad and copy and paste the text into the SQL Statement area. Then click **Next**.
   e. On the Configure the Flat File Destination page, click **Edit Mappings**, review the data, and then click **OK**. Click **Next**.
   f. On the Save and Run Package page, click **Next**.
   g. Click Finish.
   h. Review the results. **278 rows** should be transferred.
3. Use Notepad or Excel to review the csv file that was created.
4. We are now going to use the flat file that we created in Step 3 to import the data from the **\Student Files****EmployeeAccountsOpened.csv** and create a new table in the RetailBankingSample database using this data. The table will be called **EmployeeAccountsOpened** and will include all columns and rows from the CSV file.
   a. In Object Explorer, right-click the RetailBankingSample database, and then click **Tasks | Import Data**.
   b. On the Choose a Data Source page, select **Flat File Source**. Browse and locate the **\Student Files\EmployeeAccountsOpened.csv**. Switch the extension drop-down to CSV files (*.csv) to see the EmployeeAccountsOpened file. Click on Columns on the left side to review the columns that were imported. Click **Next**.
   c. On the Choose a Destination page, select **SQL Server Native Client 11.0** in the Data Source drop-down and verify that the Database is set to RetailBankingSample before clicking **Next**.
   d. On the SELECT Source Tables and Views, notice that the destination table will have the same name as the source file, and then click **Edit Mappings**.

Notice that Create destination table is selected and that all of the data types will be varchar (50) and then click **OK**. Click **Next**.

e. On the Save and Run Package page, click **Finish**.
f. Click **Finish**. Review the results. **278 rows** should have been transferred.

5. Review the contents of the new **EmployeeAccountsOpened.** A refresh may be necessary.

 Copyright © 2018 Ann L. Weber

Chapter 2 - The SELECT Statement

In this chapter:

The SELECT Statement
Execution Order of SELECT Statements
Expressions
Ordering Results
Filtering Rows
Comparison Operators
Logical Operators
Additional SELECT Options
Chapter 2 Lab
Answers to Exercises

Files needed:

- \Chapter 02 SELECT\Inline Samples
- \Chapter 02 SELECT\Try It Exercises
- \Chapter 02 SELECT\Labs\
- \Student Files

	Some of the Try It exercises in this chapter build on one another. They independent of other chapters. Completed Try It queries can be found in the \Chapter 02 SELECT \Try It Exercises folder.

The SELECT Statement

One of the most important statements in SQL is the SELECT query. The purpose of a SELECT query is to retrieve data that's already in your database. The SELECT statement returns the requested data in a rowset. Each row represents a record. The columns are the fields that hold the data associated with that record.

The SELECT and FROM Clauses

The SELECT clause defines the columns that will be returned. Within a SELECT statement you must include or one or more columns, one or more expressions, or an asterisk (*) which returns all columns. Most queries also include a FROM clause that defines the table, view, or function where the columns reside.

Syntax

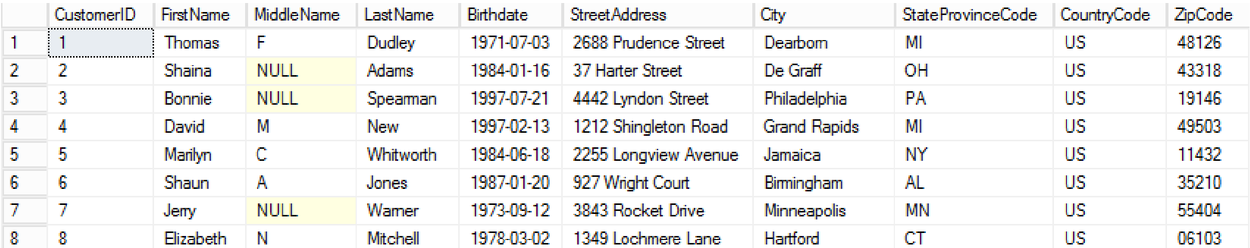
```
SELECT [ALL|DISTINCT]
[TOP (expression) [PERCENT] [WITH TIES]]
    {
      *
      |{table_name|view_name|table_alias}.*
      |{column_name|expression}
         [[AS] column_alias]
      |column_alias = expression
    } [,...n]
[FROM table_name|view_name|derived_table_definition]
[ WHERE <search_condition> ]
[ GROUP BY <group_by_clause> ]
[ HAVING <search_condition> ]
[ ORDER BY <order_by_expression> ]
[...]
```

The syntax diagrams in this course have been simplified to show only the portions relevant to the current topic.
For a full description of Microsoft SQL Syntax Conventions, see https://docs.microsoft.com/en-us/sql/t-sql/language-elements/transact-sql-syntax-conventions-transact-sql?view=sql-server-2017.

 Copyright © 2018 Ann L. Weber

Samples

```
SELECT * FROM dbo.Customer;
```



	CustomerID	FirstName	MiddleName	LastName	Birthdate	StreetAddress	City	StateProvinceCode	CountryCode	ZipCode
1	1	Thomas	F	Dudley	1971-07-03	2688 Prudence Street	Dearborn	MI	US	48126
2	2	Shaina	NULL	Adams	1984-01-16	37 Harter Street	De Graff	OH	US	43318
3	3	Bonnie	NULL	Spearman	1997-07-21	4442 Lyndon Street	Philadelphia	PA	US	19146
4	4	David	M	New	1997-02-13	1212 Shingleton Road	Grand Rapids	MI	US	49503
5	5	Marilyn	C	Whitworth	1984-06-18	2255 Longview Avenue	Jamaica	NY	US	11432
6	6	Shaun	A	Jones	1987-01-20	927 Wright Court	Birmingham	AL	US	35210
7	7	Jerry	NULL	Warner	1973-09-12	3843 Rocket Drive	Minneapolis	MN	US	55404
8	8	Elizabeth	N	Mitchell	1978-03-02	1349 Lochmere Lane	Hartford	CT	US	06103

Figure 8: Partial Results Set

```
SELECT FirstName, LastName, CustomerID FROM dbo.Customer;
```

	FirstName	LastName	CustomerID
1	Thomas	Dudley	1
2	Shaina	Adams	2
3	Bonnie	Spearman	3
4	David	New	4
5	Marilyn	Whitworth	5
6	Shaun	Jones	6
7	Jerry	Warner	7
8	Elizabeth	Mitchell	8

Figure 9: Partial Results Set

Although semi-colons (;) were not required at the time of publication for all commands, certain commands require that the previous statement in the batch end with a semi-colon. Additionally, Microsoft has stated that semi-colons will be required in a future version. Thus, it is a good practice to start using semi-colons now. This book will use semi-colons at the end of all SQL statements.

Although SELECT * FROM *tablename* is the easiest SELECT statement to write, it should be avoided to optimize the performance and readability of your queries.

Fully Qualified Object Names

When you are referencing objects in a database, such as tables, columns, and stored procedures, you can use either a partial or a fully qualified name depending on the current context and situation.

The fully qualified context is assembled as follows:

```
ServerInstanceName.DatabaseName.SchemaName.ObjectName
```

Sample

```
SQL1.RetailBankingSample.dbo.Customer
```

In the sample above, the SQL Server instance name is SQL1, while RetailBankingSample is the database name. If SQL Server was installed as a named instance named Instance1 on the server named SQL1, the fully qualified name would be entered as follows:

```
[SQL1\Instance1].RetailBankingSample.dbo.Customer
```

	Square brackets are required around the Server name when referencing a named instance because the backslash (\) is an invalid character in T-SQL.

When you open or create a query in SQL Server Management Studio, a connection is made to a SQL Server instance. Additionally, the database context is set to either the user's default database or the currently active database, depending on how the new query is created.

Once connected, you can change the database context by executing the USE *DatabaseName* command or by selecting a different database in the drop-down list in the SQL Editor toolbar as shown in Figure 10.

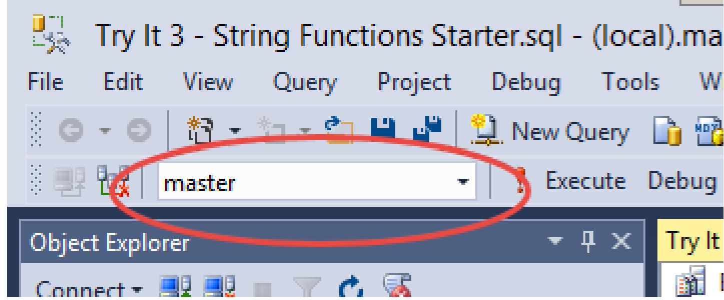


Figure 10: Database Selection Dropdown

Although older versions of SQL Server defaulted to double-quotes " " as object delimiters, you should use square brackets [ ] around any names that include reserved words or illegal characters. It is never a problem to include the square brackets around all object names as shown below:

```
[SQL1\Instance1].[RetailBankingSample].[dbo].[Customer]
```

Partially Qualified Naming

Depending on the current context, you can skip one or more parts as shown in the following three samples.

Samples

```
RetailBankingSample..Customer (the two periods allow you to
skip the schema)

dbo.Customer
```

 Copyright © 2018 Ann L. Weber

```
Customer
```

	The databases used in this course all employ the dbo schema. For simplicity's sake, most of the table name references in this course will use the one-part object names.

	For optimal performance and to remove a chance of accidentally returning the wrong data, schema names should always be included in object definitions and stored scripts. Depending on the situation, database and server names may also be recommended. Starting with SQL 2005, the concept of a schema replaced the concept of ownership for determining the context of the object. The dbo schema was created to facilitate backwards compatibility. To support this, when the query analyzer encounters an object name that does not reference a schema it will first look in the default schema for the user running the query, and then look in the dbo schema. If the object name cannot be resolved, SQL Server will return an error. Because the same object name can exist in multiple schemas, adding the schema name to all references avoids ambiguity and the possibility of retrieving the wrong data. This is similar to having two unique files on your hard drive with the same name, but located in different folders.

Using Column and Table Aliases

SQL Server allows you to use aliases as reference names for both tables and columns, although the rules for using table aliases differ from those for column aliases. Once a table name is aliased in the FROM clause, the alias must be used throughout the query rather than the actual table name. You can still use column names without referencing the table name at all, but you can no longer specify the full table name once the alias is defined.

Also, because the FROM clause is processed first, tables aliases can be used throughout all parts of the query. Column aliases, on the other hand, can be used only in the ORDER BY clause.

	When you define an alternate name for a column, the result sets using an ORDER BY based on the original column name vs result sets using the new alias name will produce the same results. The same is true for referring to a derived column with its formula vs the alias name. On the other hand, the ordering may be different when you sort on the original column name used in a derived column vs the alias or derived column formula. Even a CONVERT statement can change the sort order because a numeric field will sort 1,2,3,4,5,6,7,8,9,10,11 while a character field will sort 1,10,11,2,3,4,5,6,7,8,9.

Syntax

Table alias:

```
TableName [AS] Alias
```

Column alias:

```
ColumnName [AS] Alias
```

or

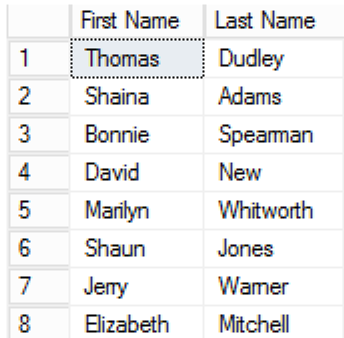
```
Alias = ColumnName
```

Most query writers find the first column alias option easier to read and understand. Many people recommend including the optional AS, especially for column aliases, to allow readers to recognize that the query writer intended to create an alias rather than simply forgetting a comma between columns. In the example below, square brackets [] are used around the alias names because of the space inside the column alias.

Sample

```
SELECT C.FirstName AS [First Name], C.LastName AS [Last Name]
FROM Customer AS C
;
```

 Copyright © 2018 Ann L. Weber



	First Name	Last Name
1	Thomas	Dudley
2	Shaina	Adams
3	Bonnie	Spearman
4	David	New
5	Marilyn	Whitworth
6	Shaun	Jones
7	Jerry	Warner
8	Elizabeth	Mitchell

Figure 11: Partial Results Set

IntelliSense, like the SQL analyzer, reads the FROM clause first. To improve the result lists that IntelliSense presents, write your FROM clause first and then go back and write all the columns in the SELECT clause. Adding a table alias and then referencing it in the SELECT clause followed by a period "."provides you with a list of columns in that table.

Execution Order of SELECT Statements

To better understand why certain rules exist when writing and executing queries, it is important to understand the order in which the SQL Server engine interprets the SELECT statement.

Although the syntax requires that you define the columns to be returned immediately following the word SELECT, this is one of the last parts of the statement to actually be analyzed and executed. The following list displays the syntax order, while the numbers represent the execution order.

6 - SELECT

1 - FROM

2 - JOIN

3 - WHERE

4 - GROUP BY

5 - HAVING

7 - ORDER BY

When you realize that the ORDER BY clause is the only part of the query to run after the SELECT, it explains why column aliases can be referenced in an ORDER BY clause, but not in the HAVING or WHERE clauses.

Try It 1 – Basic SELECT Statement

In this exercise, you will create a new Query Editor window, save the query file, and then write and execute a basic query that you will build on in later Try It exercises.

1. If necessary, open SQL Server Management Studio (SSMS).
2. Click the New Query button in the General toolbar to open a new Query Editor tab.
3. Click **File** | **Save** (or click the Save icon). Browse to the **\Student Files** folder, change the File name to **BasicSELECT.sql**, and then click **Save**.
4. Either select RetailBankingSample from the database drop-down list in the SQL Editor toolbar, or type and execute the following SQL command, and then Press F5 or click the Execute button to execute the script.

```
USE RetailBankingSample;
```

5. Type the following command to retrieve the LoanTransactionID, TransactionDate, and Amount columns from the LoanTransaction table. The table name will be aliased as LT.

Remember, if you type the FROM line first, then go back to the SELECT line and use the alias "LT." at the beginning of each column name, IntelliSense will help you type in your query. If you are efficient at typing, simply typing in the query may be more efficient for you.

```
SELECT LT.LoanTransactionID, TransactionDate, Amount
FROM LoanTransaction AS LT
;
```

6. Execute the query. **21,450 rows** will be returned and the result set will look similar to Figure 12.

	LoanTransactionID	TransactionDate	Amount
1	1	2007-05-29 00:00:00	186.89
2	2	2007-06-29 00:00:00	186.73
3	3	2007-07-29 00:00:00	186.57
4	4	2007-08-29 00:00:00	186.41
5	5	2007-09-29 00:00:00	186.25
6	6	2007-10-29 00:00:00	186.08
7	7	2007-11-29 00:00:00	185.92
8	8	2007-12-29 00:00:00	185.75

Figure 12: Partial Results Set

7. Save your query and leave the query tab and SSMS open for the next Try It exercise.

 Copyright © 2018 Ann L. Weber

Expressions

In addition to returning individual columns, the SELECT statement accepts expressions. These expressions can contain string literals, columns concatenated together, mathematical operations, and much more.

String Literals

SQL allows you to incorporate strings into your query results either as a standalone column or concatenated in with other columns. All string values need to be enclosed in single quotes (' ').

Sample

```
SELECT 'Credit Transaction' AS [Transaction Source]
   , TransactionDate
   , Amount
FROM CreditTransaction
;
```

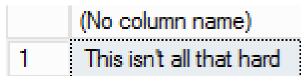
	Transaction Source	TransactionDate	Amount
1	Credit Transaction	2008-06-24 00:00:00.0000000	-351.94
2	Credit Transaction	2009-04-26 00:00:00.0000000	-579.90
3	Credit Transaction	2011-11-14 00:00:00.0000000	-869.67
4	Credit Transaction	2015-09-10 00:00:00.0000000	-126.59
5	Credit Transaction	2007-01-22 00:00:00.0000000	-281.88
6	Credit Transaction	2008-08-30 00:00:00.0000000	-1392.51
7	Credit Transaction	2015-11-18 00:00:00.0000000	-283.08
8	Credit Transaction	2014-04-29 00:00:00.0000000	-174.67

Figure 13: Partial Results Set

If you want to include a single quote in the data, you must “escape” the single quote by placing another single quote immediately before it.

Sample Escape Sequence

```
SELECT 'This isn''t all that hard';
```



	(No column name)
1	This isn't all that hard

Figure 14: Results

You will learn more about working with string expressions in **Chapter 3 - Built-In Functions Overview**.

Concatenation

When working with string data types you can combine multiple fields using a technique called concatenation. SQL provides two methods to concatenate data.

The CONCAT function accepts multiple expressions as parameters that will be joined together into a single field. Alternatively, you can use a + sign to concatenate columns, strings, or expressions together.

When you combine columns or expressions and either side of the plus sign (+) resolves to NULL, the new expression will resolve as NULL. On the other hand, the CONCAT function ignores any NULL values, replacing them with an empty string. You will learn more about working with NULL values and the CONCAT function in **Chapter 4 Handling NULL Data**.

Sample

```
SELECT FirstName + ' ' + LastName AS [Full Name]
FROM Customer;
```

	Full Name
1	Thomas Dudley
2	Shaina Adams
3	Bonnie Spearman
4	David New
5	Marilyn Whitworth
6	Shaun Jones
7	Jerry Warner
8	Elizabeth Mitchell

Figure 15: Partial Results Set

Concatenation works with string data. When you mix data types with one side of the concatenation symbol having a character data type and the other side with a numeric data type, SQL will treat the plus sign as addition and will try to convert your characters to numbers. You can avoid this by converting your numeric fields to strings before the concatenation. You will learn more about explicit and implicit data type conversions in **Chapter 3 Built-in Functions Overview**.

 Copyright © 2018 Ann L. Weber

Try It 2 – Concatenation and Literals

In this Try It exercise you will extend the query that you wrote in the first Try It exercise by concatenating together the words "Credit transaction of type " and the TransactionType into a single column aliased as TransactionTable&Type. A partial result set is displayed in Figure 16.

	LoanTransactionID	TransactionDate	Amount	Type Information
1	1	2007-05-29 00:00:00	186.89	Credit transaction of type interest
2	2	2007-06-29 00:00:00	186.73	Credit transaction of type interest
3	3	2007-07-29 00:00:00	186.57	Credit transaction of type interest
4	4	2007-08-29 00:00:00	186.41	Credit transaction of type interest
5	5	2007-09-29 00:00:00	186.25	Credit transaction of type interest
6	6	2007-10-29 00:00:00	186.08	Credit transaction of type interest
7	7	2007-11-29 00:00:00	185.92	Credit transaction of type interest
8	8	2007-12-29 00:00:00	185.75	Credit transaction of type interest

Figure 16: Partial Results Set

1. If the **BasicSELECT.sql** file is not open from the previous Try It exercise, click **File | Open | File** (or click the Open File icon) and browse to the **\Student Files\ BasicSELECT.sql** file.
   Note: If you did not complete the previous Try It exercise, browse to **\Chapter 02 SELECT\Try It Exercises\ Try It 2 - Concatenation Starter.sql**.

2. Either select RetailBankingSample from the database drop-down list in the SQL Editor toolbar, or type and execute the following SQL command. Press F5 or click the Execute button to execute the script.

```
USE RetailBankingSample;
```

3. Modify the query to match the query below. This query adds a new column to the result set that concatenates the words "Credit transaction of type " to the TransactionType column and aliases the column as **Type Information**. Be sure to add a space before the close single quote to make the results more readable.

```
SELECT LT.LoanTransactionID, TransactionDate, Amount
     , 'Credit transaction of type ' + TransactionType AS
[Type Information]
FROM LoanTransaction AS LT
;
```

4. Execute the query. **21,450 rows** should be returned, and the results should look similar to Figure 16.

5. If you are working in the BasicSELECT.sql file that you created, click Save. If you are working in the **Try It 2 - Concatenation Starter.sql** file, click File | Save … As …, and then save the script to your \Student Files folder.
6. Close the current query tab, but leave SSMS open for the next Try It exercise.

Arithmetic Expressions

SQL provides the standard arithmetic operators to allow you to create expressions based on the data in numerical columns and values. Although everyone should easily recognize the first four operators in Table 1 below, modulus may be new to some query writers. Modulus provides the remainder when a value in a column is divided by another value. This can be used to determine odd or even numbers.

Table 1: Arithmetic Expressions

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)

Syntax Modulus

```
dividend % divisor
```

Returns the remainder of the first number divided by the second.

Sample Modulus

```
SELECT 10 % 5 AS Remainder;
```

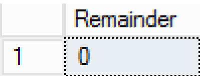


Figure 17: Results

```
SELECT 12 % 5 AS Remainder;
```

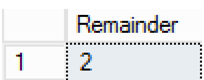


Figure 18: Results

 Copyright © 2018 Ann L. Weber

Try It 3 – Arithmetic Expressions

In this exercise you will practice using the Modulus operator and modify an existing query to determine if the total number of customers in the Customer table is evenly divisible by 5. You will then modify an existing query to add .3% (or .003) to all savings accounts to see what these new values would be. Savings accounts have an AccoutTypeID of 2 or 5.

1. Click the **Open File** icon or File | **Open** | **File**.
2. Browse to the **\Chapter 02\Try It Exercises** folder, and then open the **Try It 3 - Arithmetic Expressions Starter.sql** file.
3. Click **File** | **Try It 3 - Arithmetic Expressions Starter.sql As**, and then browse to **the \Student Files** folder. Type **Ch2Ti3.sql** in the File name, and then click **Save**.
4. Either select RetailBankingSample from the database drop-down list in the SQL Editor toolbar, or highlight and execute the following SQL command. Press **F5** or click the **Execute** button to execute the script.

```
USE RetailBankingSample;
```

5. Highlight and execute the query under Step #5. The count should be 300.
6. Modify the query under Step #4 as follows to divide the total customer count by 5 return the remainder. Change the alias of this column to Remainder. The remainder should be 0 as 300 is divisible by 5.

```
SELECT  COUNT(*) % 5 AS Remainder
FROM Customer
;
```

7. Highlight just the query under Step #7 and execute the query. Review the data.
8. Modify and then execute this query to create a new derived column named ProposedInterestRate. The value in this new column should be the current value plus .3%. A sample query and partial result set are provided below.

```
SELECT AccountID, AccountTypeID, PrimaryCustomerID
     , InterestRate AS CurrentInterestRate
     , InterestRate + .003 AS ProposedInterestRate
FROM Account AS A
WHERE AccountTypeID IN (2,5)
;
```

	AccountID	AccountTypeID	PrimaryCustomerID	CurrentInterestRate	ProposedInterestRate
1	2	2	109	0.005	0.008
2	4	5	132	0.014	0.017
3	7	5	229	0.014	0.017
4	19	5	187	0.014	0.017
5	21	5	17	0.014	0.017
6	22	2	122	0.005	0.008
7	32	2	166	0.005	0.008
8	35	2	218	0.005	0.008

Figure 19: Partial Results Set

9. Save and then close your query. Leave SSMS open for the next Try It exercise.

Working with CASE Expressions

Although you can't use control of flow operators such as IF ELSE within the context of a SELECT statement, the CASE expression provides a similar functionality to the IF ELSE key words.

Starting with SQL 2012 you can optionally use IIF to provide functionality similar to a basic CASE or IF. You will learn more about built-in functions in **Chapter 3 Using Built-in Functions**.

Simple CASE

Although some would argue that there isn't anything "simple" about a case statement, this is, nevertheless, the name of this first type of CASE. You can tell that a CASE is simple because the column name is only referenced once near the beginning of the expression. The WHEN clause of the statement is therefore "simple". Rather than specifying both side of the condition, such as color = 'red', you would simply type 'red' within the WHEN. The column name comes earlier in the statement and the equal operator is assumed. For this reason, if you need to test an inequality, check for nulls, combine two conditions, or other more complex tests, you must use a searched case.

You can think of a WHEN clause as a bucket that catches any rows that match the condition. If a row isn't caught by the first bucket, it will continue down through the conditions until it finds a match.

Syntax

```
CASE input_expression
     WHEN when_expression THEN result_expression [ ...n ]
     [ ELSE else_result_expression ]
END
```

Sample

```
SELECT LT.LoanTransactionID, Amount
    , CASE TransactionType
          WHEN 'Interest' THEN 'Int'
          WHEN 'MonthlyPayment' THEN 'Pmt'
```

 Copyright © 2018 Ann L. Weber

```
    END AS TypeAbbr
FROM LoanTransaction AS LT
;
```

	LoanTransactionID	Amount	TypeAbbr
1	1	186.89	Int
2	2	186.73	Int
3	3	186.57	Int
4	4	186.41	Int
5	5	186.25	Int
1...	11418	-195.35	Pmt
1...	11419	-195.35	Pmt
1...	11420	-195.35	Pmt

Figure 20: Partial Results Set

Searched Case

The second type of CASE allows much greater flexibility. With a searched case, the WHEN clause contains comparisons, and the THEN clause contains the information to be returned. Where a simple case can only check equality to a single field, the searched CASE can compare two fields, compare a single field to a list of optional values, compare to a range of values and more. Another key benefit of a searched CASE is the ability to test for NULL values. A simple case cannot test for NULLs.

A searched CASE does require more typing because you must define the column or expressions to be compared in each WHEN clause.

Syntax

```
CASE
     WHEN Boolean_expression THEN result_expression [ ...n ]
     [ ELSE else_result_expression ]
END
```

Sample

```
SELECT E.EmployeeID
   , CASE
            WHEN E.HireDate > '20160101' THEN 'Recent Hire'
            ELSE 'Veteran Employee'
       END AS EmploymentLength
FROM Employee AS E
;
```

	EmployeeID	EmploymentLength
1	1	Veteran Employee
2	2	Veteran Employee
3	3	Veteran Employee
4	4	Veteran Employee
5	5	Recent Hire
6	6	Veteran Employee
7	7	Veteran Employee
8	8	Veteran Employee

Figure 21: Partial Results Set

ELSE

If you do not provide an ELSE clause, any value that does not match one of the WHEN clauses will return NULL. Generally, you should include an ELSE statement to collect unexpected data. For example, the CreditTransaction table includes four values in the TransactionType column. If you write the case expression to replace each of the values with a new three-character abbreviation, you should define all 4 values in a WHEN clause, and use the ELSE clause for something similar to “Unsupported Value” rather than using the ELSE to catch the fourth value or not specifying an ELSE clause at all.

Sample

```
SELECT CT.CreditTrxID, Amount
   , CASE TransactionType
        WHEN 'Interest' THEN 'Int'
        WHEN 'Payment' THEN 'Pmt'
   ELSE 'na'
   END AS TypeAbbr
FROM CreditTransaction AS CT
;
```

	CreditTrxID	Amount	TypeAbbr
1	1	-351.94	na
2	2	-579.90	na
3	3	-869.67	na
4	4	-126.59	na
5	5	-281.88	na
8...	84542	-54.12	Int
8...	84543	-0.04	Int
8...	84544	-0.14	Int
8...	84545	10709....	Pmt
8...	84546	17363....	Pmt
8...	84547	8797.24	Pmt

Figure 22: Partial Results Set

 Copyright © 2018 Ann L. Weber

<table>
<tr>
<td>
</td>
<td>If written carefully, order of the WHEN clauses does matter. To optimize performance, the WHEN clause that will match the most rows should go first, continuing until the smallest matching row count is last.

In SQL, as soon as a match is made in a CASE statement, the SQL processor drops out of the WHEN clause and moves on to the next row and does not continue through the other WHEN clauses with that value. This feature allows you to write less complex comparisons. For example, if the first “bucket” you are checking for in your CASE statement is less than 5, you don’t have to define the second bucket as greater than 5 and less than 10. You can simply define the second “bucket” as less than 10. Every row with the CASE value less than 5 has already been caught by the first “bucket.”</td>
</tr>
</table>

Try It 4 – Simple CASE

In this exercise you will write a simple CASE expression to return a new abbreviation column with a shortened version of the transaction types in the Account Transaction table. You will create an option for each type and add a warning in the ELSE clause that warns the user that a new transaction type may have been added. You will practice a searched CASE expression in the lab, after you have learned about comparison operators. A partial result set is shown in Figure 23.

	AcctID	AccountTransactionID	Amount	(No column name)
1	12	1	2978.15	MA
2	12	2	-1135.97	ATM
3	12	3	3991.69	DD
4	12	4	-2817.13	ATM
5	12	5	2357.13	DD
6	12	6	1043.05	ATM
7	12	7	-1830.80	MA
8	12	8	-1542.84	DC

Figure 23: Partial Results Set

1. Click the New Query button in the General toolbar to open a new Query Editor tab.
2. Click **File | Save** (or click the Save icon). Browse to the **\Student Files** folder, change the File name to **SimpleCase.sql**, and then click **Save**.
3. Either select RetailBankingSample from the database drop-down list in the SQL Editor toolbar, or type and execute the following SQL command. Press F5 or click

the Execute ! button to execute the script.

```
USE RetailBankingSample;
```

4. Write and execute a query that will return the AcctID and Amount fields from the AccountTransaction table.
5. Add a third column aliased as TypeAbbr. The column should return the following abbreviations based on each TransactionType value in the table. A sample query is below the list.
    a. In Bank – IB
    b. Debit Card – DC
    c. Direct Deposit – DD
    d. Mobile App – MA
    e. ATM – ATM
    f. Check - CHK

```
SELECT AT.AcctID, AT.AccountTransactionID,AT.Amount
     , CASE AT.TransactionType
          WHEN 'In Bank' THEN 'IB'
          WHEN 'Debit Card' THEN 'DC'
          WHEN 'Direct Deposit' THEN 'DD'
          WHEN 'Mobile App' THEN 'MA'
          WHEN 'ATM' THEN 'ATM'
          WHEN 'Check' THEN 'CHK'
     ELSE 'A new type may have been added to the
database. Please check with your DBA'
          END
FROM AccountTransaction AS AT
;
```

6. Execute your query and verify the results. Because every row matches one of the WHEN clauses, no rows should return your warning about a new transaction type.
7. Save your query and close the current query tab, leaving SSMS open for the next Try It exercise.

Ordering Results

If you want to guarantee the order of the rows in your result set, you must add an ORDER BY clause to your query. In the absence of an ORDER BY clause, the server displays the results in the order the data was in when the processing was finished. Because most simple queries use the same execution plans every time they run, query writers sometimes wrongly assume that the data is guaranteed to be returned in the order that it is stored in the table, or some other manner, but this is not true.

The ORDER BY clause is frequently the very last clause in a SQL statement.

 Copyright © 2018 Ann L. Weber

Syntax

```
ORDER BY order_by_expression
    [ COLLATE collation_name ]
    [ ASC | DESC ]
    [ ,...n ]
```

Sample

```
SELECT C.CustomerID, C.FirstName, C.LastName
FROM Customer AS C
ORDER BY C.LastName, C.FirstName
;
```

If the ORDER BY clause does not specify ascending (ASC) or descending (DESC) order, the default is ascending. Each column in the sort clause has an independent sort order. If you change C.LastName to DESC in the example above, the last name will be sorted Z to A, but the first name will still be sorted A to Z.

Try It 5 – Sorting Result Sets

In the following exercise you will write a query to return the CustomerID, City, and StateProvinceCode fields for each customer. Sort the result so that the states are sorted in reverse alphabetical order, but the cities are sorted from A to Z within each state. A partial result set is shown in Figure 24.

	CustomerID	City	StateProvinceCode
1	287	Worland	WY
2	171	Appleton	WI
3	23	Appleton	WI
4	35	Milwaukee	WI
5	138	Milwaukee	WI
6	164	Sturgeon Bay	WI
7	189	Ashford	WA
8	259	Bellevue	WA

Figure 24: Partial Results Set

1. Create a new query window and save your query to the **\Student Files** folder as **Sorting.sql**. If you need help you can refer back to the first Try It exercise in this chapter.
2. Set the database context to **RetailBankingSample**.
3. Write and execute a query to return the **CustomerID**, **City**, and **StateProvinceCode** fields for each customer. Sort the results so that the states are sorted in reverse alphabetical order, but the cities are sorted from A to Z within each state. The sample code is included below.

```
SELECT C.CustomerID, C.City, C.StateProvinceCode
FROM Customer AS C
```

Copyright © 2018 Ann L. Weber

```
ORDER BY C.StateProvinceCode DESC, C.City ASC
;
```

4. Save your query and close the query tab. Leave SSMS open for the next Try It exercise.

Filtering Rows

So far, the queries you've been looking at return all the rows in the tables listed in the FROM clause. When at all possible, your queries should include WHERE clauses to filter the data. This will improve query performance, especially with proper indexing, and also returns fewer rows to the users and analysts that need to review the information.

The WHERE clause goes between the FROM and ORDER BY clauses when writing the query. You do not need to include the search condition columns in the SELECT clause, although you may want to do this at least temporarily to make verification easier.

Syntax

```
WHERE <search_condition>

<search_condition> ::=
    {[NOT] <predicate>|(<search_condition>)}
    [{AND|OR}[NOT]{<predicate>|(<search_condition>)}]
[,...n]

<predicate> ::=
   {  expression {=|<>|!=|>|>=|!>|<|<=|!<} expression
    | string_expression [NOT] LIKE string_expression
    | expression [NOT] BETWEEN expression AND expression
    | expression IS [NOT] NULL }
```

There are many operators and options for the search conditions available within WHERE clauses.

Comparison Operators

The first set of operators will seem familiar to most everyone. They are the comparison operators listed in the Table 2 below.

Table 2: Comparison Operators

Operator	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

 Copyright © 2018 Ann L. Weber

	Although you may see existing code that uses the exclamation point (! or "bang" in Unix terminology) to mean NOT, such as "!=" for not equal to, this is not an ISO standard. In later versions of SQL, the query optimizer translates !< and !> to their equivalent >= or <= before processing to make the comparison SARGable. A SARGable query is one where an index seek can be used.You will learn how create indexes in **Chapter 10 Data Definition Language**.

Comparison operators can be used with all expressions except those that are defined as the text, ntext, or image data types. These data types have been deprecated and should be removed from databases when possible.

Numeric comparisons are probably the most straightforward thanks to elementary math classes: 1 is less than 2 and 3 is greater than 2.

Sample Numeric

```
SELECT AT.AccountTransactionID, AT.Amount
FROM AccountTransaction AS AT
WHERE AT.Amount >= 5000
;
```

	AccountTransactionID	Amount
1	54	5451.00
2	71	5426.12
3	257	5302.96
4	258	5080.25
5	263	6759.97
6	295	5494.52
7	455	5375.21
8	467	7751.55

Figure 25: Partial Results Set

Where things start to get a bit trickier is when we are working with characters and dates.

Dates are entered as strings and then stored as a number offset from a defined time. How this offset works varies depending on the specific data type. There are many supported string formats including the following:

- `'March 8, 2019'`
- '20190308' (zeros are required – must be 6 or 8 digits)
- '190308' (zeros are required – must be 6 or 8 digits)
- `'3/8/2019'`
- '2004-03-08' (zeros are optional)

When specifying a two-digit year, values up to and including 49 are interpreted as 20xx. 50 and above are interpreted as 19xx. It is recommended that you always specify four-digit years when saving scripts for future use.

This book will typically use the ‘yyyymmdd’ format for strings that will be interpreted as dates.

Sample Date

```
SELECT AT.AccountTransactionID, AT.TransactionDate
FROM AccountTransaction AS AT
WHERE TransactionDate = '20180101'
;
```

	AccountTransactionID	TransactionDate
1	3019	2018-01-01 00:00:00
2	3307	2018-01-01 00:00:00
3	8137	2018-01-01 00:00:00
4	8950	2018-01-01 00:00:00
5	19973	2018-01-01 00:00:00
6	24865	2018-01-01 00:00:00
7	28211	2018-01-01 00:00:00
8	29072	2018-01-01 00:00:00

Figure 26: Partial Results Set

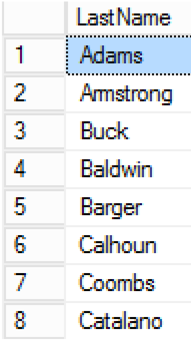
If you would like to know more about how dates are stored and even more about what happens inside of SQL Server see Kalen Delaney’s SQL Server 2012 Internals book. Even though the print version is no longer available, it is still very relevant and the ebook version is available at https://www.microsoftpressstore.com/store/microsoft-sql-server-2012-internals-9780735658554 or from Amazon as a Kindle book.

Searching character fields can be a little less intuitive, especially when using a case sensitive database or custom collations. Some comparisons are obvious, such as A is less than B and C is greater than B. But as we saw earlier with ordering numbers that are stored as characters, '10' is less than '2'. Also, 'CA' is greater than 'C' and less than 'CAA'.

Sample Character

```
SELECT C.LastName
FROM Customer AS C
WHERE C.LastName < 'D'
;
```

 Copyright © 2018 Ann L. Weber



	LastName
1	Adams
2	Armstrong
3	Buck
4	Baldwin
5	Barger
6	Calhoun
7	Coombs
8	Catalano

Figure 27: Partial Results Set

Try It 6 – Comparison Operators

In this Try It exercise, you will write a query that will return only Accounts with an opening date on or after January 1, 2012. You will also write a query to return accounts with an opening balance over $10,000. A partial result set of the first query is shown in Figure 28.

	AccountID	OpeningDate	OpeningBalance
1	4	2015-10-25	37048.13
2	9	2015-05-31	32963.78
3	14	2017-08-23	44935.32
4	17	2013-03-26	34946.31
5	19	2013-09-23	26882.97
6	20	2014-07-16	3319.25
7	25	2014-01-16	33995.04
8	32	2012-10-04	32265.69

Figure 28: Partial Results Set

1. Create a new query window and save your query to the **\Student Files** folder as **Comparison Operators.sql**. If you need help you can refer back to the first Try It exercise in this chapter.
2. Set the database context to **RetailBankingSample**.
3. Write and execute a query to return the **AccountID**, **OpeningDate**, and **OpeningBalance** from the Account table that restricts the rows to only those with an **OpeningDate** on or after **January 1, 2012**. **98 rows** should be returned. The query is shown below.

```
SELECT A.AccountID, A.OpeningDate, A.OpeningBalance
FROM Account AS A
WHERE A.OpeningDate >= '20120101'
;
```

4. Write a second query that returns the same columns, but only returns rows with an OpeningBalance of more the $10,000 as shown in Figure 29. **203 rows** should be returned. The query is shown below.

```
SELECT A.AccountID, A.OpeningDate, A.OpeningBalance
FROM Account AS A
WHERE A.OpeningBalance > 10000
;
```

	AccountID	OpeningDate	OpeningBalance
1	1	2007-03-29	33009.37
2	2	2009-06-13	39800.27
3	4	2015-10-25	37048.13
4	5	2010-06-19	17185.94
5	6	1985-03-31	34732.16
6	7	1988-02-02	44147.88
7	8	2006-07-15	43654.10
8	9	2015-05-31	32963.78

Figure 29: Partial Results Set

5. Save your query, but leave it and SSMS open for the next Try It exercise.

Logical Operators

Like comparison operators, logical operators return TRUE, FALSE, or UNKNOWN. Logical operators test to see if one or more conditions exist. The following table summarizes the behavior of each operator.

Table 3: Logical Operators

AND	TRUE if both Boolean expressions are TRUE.
BETWEEN	TRUE if the operand is within the inclusive range.
IN	TRUE if the operand is equal to one of a list of expressions.
LIKE	TRUE if the operand matches a pattern.
NOT	Reverses the value of any other Boolean operator.
OR	TRUE if either Boolean expression is TRUE.

It is extremely important to pay attention to the order of operations with all operators including arithmetic, comparison, and logical. Parentheses can be used to guarantee execution order both when you are unsure about the order of operations and when you want to override the order of operations.

Many years ago, I had a student that was trying to retrieve a set of computers with certain characteristics to be able to upgrade to Windows 98. (Yes, I said a long time ago.) Because they did not understand the order of operations between the logical and comparison operators that they were using, they continually retrieved the wrong set of computers. When he brought the query into class, we added a few sets of parentheses and were then able to retrieve the correct set of computers.

 Copyright © 2018 Ann L. Weber

AND/OR/NOT

When writing a list of comparisons in a WHERE clause, you must include the column name for each comparison. You have the flexibility to perform comparisons that refer to any of the columns from the table(s) in the FROM clause, not just the columns included in the SELECT.

AND requires all conditions connected with AND to evaluate to TRUE before the final answer is TRUE. OR only requires one condition to evaluate to TRUE for it to send a TRUE response. NOT negates whatever operation is happening.

You can mix any number of operators and comparisons, but be careful to fully understand the order of operations and what results will be returned.

Samples

```
SELECT C.FirstName, C.LastName
FROM Customer AS C
WHERE  LastName = 'Johnson'
   AND FirstName = 'Roy'
;
```

	FirstName	LastName
1	Roy	Johnson

Figure 30: Results

Rows are returned only if both the FirstName is Roy and the LastName is Johnson.

```
SELECT C.FirstName, C.LastName
FROM Customer AS C
WHERE  LastName = 'Johnson'
   OR FirstName = 'Roy'
;
```

	FirstName	LastName
1	Steve	Johnson
2	Kala	Johnson
3	Roy	Deleon
4	Ethel	Johnson
5	Roy	Johnson
6	Marla	Johnson

Figure 31: Results

Rows are returned if the customer has either a LastName of Johnson or a FirstName of Roy.

```
SELECT C.FirstName, C.LastName
FROM Customer AS C
WHERE NOT LastName = 'Johnson'
   AND (FirstName = 'Steve'
```

```
            OR FirstName =  'Roy')
;
```

	FirstName	LastName
1	Roy	Deleon

Figure 32: Results

Rows are returned if the FirstName is either Roy or Steve and the last name is not Johnson. There is only 1 customer named Steve in the database, but his last name is Johnson, so he is eliminated. If we added another set of parentheses before LastName and after Roy, we would get every customer except for Steve Johnson and Roy Johnson.

Try It 7 – Logical Operators

In this exercise, you will modify the queries that you wrote in Try It 6 to create a single query that only returns accounts with both an opening date on or after Jan 1, 2012 and a balance over $10,000. A partial results set in shown in Figure 33.

	AccountID	OpeningDate	OpeningBalance
1	4	2015-10-25	37048.13
2	9	2015-05-31	32963.78
3	14	2017-08-23	44935.32
4	17	2013-03-26	34946.31
5	19	2013-09-23	26882.97
6	25	2014-01-16	33995.04
7	32	2012-10-04	32265.69
8	36	2017-01-04	59156.30

Figure 33: Partial Results Set

1. If the **Comparison Operators.sql** file is not open from the previous Try It exercise, click **File | Open | File** (or click the Open File icon) and browse to the **\Student Files\ Comparison Operators.sql** file.
   Note: If you did not complete the previous Try It exercise, browse to **\Chapter 02 SELECT\Try It Exercises\ Try It 7 – Logical Operators Starter.sql**.

2. Modify the query under Step #4, add an AND after the 10000, but before the semi-colon, and then copy the WHERE clause (without the word WHERE or the semicolon) and paste it after the AND that you just added. The query should look similar to the one below. **76 rows** should be returned.

```
SELECT A.AccountID, A.OpeningDate, A.OpeningBalance
FROM Account AS A
WHERE A.OpeningBalance > 10000
     AND A.OpeningDate >= '20120101'
```

 Copyright © 2018 Ann L. Weber

```
;
```

3. Save and close your query, but leave SSMS open for the next Try It.

BETWEEN

The BETWEEN operator acts as an inclusive between two values and provides a shortcut for returning a range of values. It produces the same results as writing a greater than or equal to on one end of the condition and a less than or equal to at the other end.

Care should be taken with using BETWEEN with date and character fields. For example, BETWEEN 'a' and 'c' will return all records that start with 'a' in the defined field, including the letter 'a' by itself, but only records that are the letter 'c' by itself will be returned. The letter 'c' with any other letter, number, or special character following it is greater than 'c' and will not be returned.

Dates have a similar concern with comparisons. A date entered alone without a time is understood to be midnight of that day for comparison with datetime fields, and midnight is at the beginning of a day. If all of your data contains only dates and not times, but is stored as datetime, the BETWEEN works as expected. But once a time is entered for a datetime record, any records occurring after midnight on the outer edge of the BETWEEN range will not be included in the result set.

Syntax

```
test_expression [NOT] BETWEEN begin_expression AND
   end_expression
```

Sample with BETWEEN

```
SELECT AT.AccountTransactionID, AT.TransactionDate
FROM AccountTransaction AS AT
WHERE AT.TransactionDate
   BETWEEN '20120101' AND '20141231 23:59:59'
;
```

	AccountTransactionID	TransactionDate
1	1	2014-10-14 00:00:00
2	5	2014-03-12 00:00:00
3	9	2013-09-29 00:00:00
4	10	2012-12-19 00:00:00
5	11	2014-01-20 00:00:00
6	12	2013-10-14 00:00:00
7	14	2013-10-07 00:00:00
8	23	2013-11-10 00:00:00

Figure 34: Partial Results Set

Even though time is not typically specified in this database, writing the query as above guarantees that all records from Dec 31, 2014 will be included. If the data type in the database had a finer level of accuracy, you should continue the time field out to the defined accuracy. For example, if the data type

was datetime2(7), the time should be stated 23:59:59.9999999. An alternative to including the time is shown below.

SAMPLE with greater and less than

```
SELECT AT.AccountTransactionID, AT.TransactionDate
FROM AccountTransaction AS AT
WHERE AT.TransactionDate >= '20120101'
   AND AT.TransactionDate < '20150101'
;
```

IN

The IN operator is shorthand for multiple OR operators. The server translates the IN to a sequence of OR operations when building the execution plan, so there is minimal or no difference in performance between IN and OR. Most people find IN much easier to read and to type.

Syntax

```
test_expression [NOT] IN (subquery | expression [, …n])
```

Sample

```
SELECT TransactionType, Amount
FROM AccountTransaction
WHERE TransactionType IN ('ATM', 'Direct Deposit')
;
```

	TransactionType	Amount
1	ATM	-1135.97
2	Direct Deposit	3991.69
3	ATM	-2817.13
4	Direct Deposit	2357.13
5	ATM	1043.05
6	Direct Deposit	6.64
7	Direct Deposit	1073.20
8	Direct Deposit	855.68

Figure 35: Partial Results Set

Try to avoid NOT IN, but rather list the items for the IN phrase. With newer versions of SQL, NOT IN can perform an INDEX seek, but it translates the comparison to a series of greater than and less than statement. For example, NOT IN ('a', 'b') is interpreted as (< 'a' AND > 'a') OR (< 'b' AND > 'b'). This typically increases the logical reads on the table being read and negatively impacts performance.

 Copyright © 2018 Ann L. Weber

Try It 8 – IN and BETWEEN

In this exercise you will write a query that returns all accounts from the Account table where the AccountTypeID is any of the following: 1, 3, 4, 13, 14. Additionally, the accounts returned should have been opened sometime between January 1, 2010 and December 31, 2018. A sample result set is shown in Figure 36.

	AccountID	AccountTypeID	OpeningBalance	OpeningDate
1	109	14	0.00	2010-01-23
2	51	4	9968.81	2010-03-12
3	26	4	32899.25	2010-03-13
4	45	4	37521.90	2010-04-07
5	228	13	0.00	2010-06-08
6	97	3	57502.09	2010-09-28
7	201	13	0.00	2010-09-29
8	12	1	30765.29	2010-10-27

Figure 36: Partial Results Set

1. Create a new query window and save your query to the **\Student Files** folder as **IN and BETWEEN.sql**. If you need help you can refer back to the first Try It exercise in this chapter.
2. Set the database context to **RetailBankingSample**.
3. Write and execute a query that will return the AccountID, AccountTypeID, OpeningBalance, and OpeningDate columns from the Account table. Only accounts where the AccountTypeID is 1, 3, 4, 13, or 14 should be included. Also, only accounts opened between January 1, 2010 and December 31, 2018 should be included in the query. Use the IN operator for the AccountTypeIDs and a BETWEEN operator for the OpeningDate field. The final query should be similar to the one below. **57 rows** should be returned.

```
SELECT A.AccountID, A.AccountTypeID
     , A.OpeningBalance, A.OpeningDate
FROM Account AS A
WHERE AccountTypeID IN (1,3,4,13,14)
     AND OpeningDate BETWEEN '20100101'  AND '20181231
23:59:59'
ORDER BY OpeningDate
;
```

4. Save your query and close the query editor tab.

LIKE

The LIKE operator allows you to you to use wild cards to create very specific search patterns against string data. If any of the expressions are not strings, SQL Server will attempt to convert them to a string for the comparison.

Syntax

```
match_expression [NOT] LIKE pattern [ESCAPE escape_character]
```

Both the pattern and the escape character need to be enclosed in single quotes. Similar to wild cards in DOS or regular expressions, SQL LIKE wildcards allow you to replace one or more characters with the defined wild card character.

Percent (%)

The percent sign, like the DOS asterisk, matches zero to an infinite number of characters. For example, '%at' matches 'at', 'cat', and 'that'. It does not match 'cats'.

The following sample returns all account numbers that end with the number 75.

Sample

```
SELECT AccountNumber
FROM CustomerAccount
WHERE AccountNumber LIKE '%75'
;
```

	AccountNumber
1	DMorri75
2	BWilli75
3	EMonte75
4	SFaith175
5	MMathi275
6	JRiggs275

Figure 37: Results

Underscore (_)

The underscore (_) like the DOS question mark matches exactly 1 character. While '_at' matches 'cat' and 'hat', it does not match either 'at' or 'that'. The sample below shows a query that returns rows where the second character of the account number is an R. If a % wildcard was used at the beginning of the string, the results would have included records with an R anywhere, not just in the second character.

Sample

```
SELECT AccountNumber
FROM CustomerAccount
WHERE AccountNumber LIKE '_R%'
;
```

 Copyright © 2018 Ann L. Weber

	AccountNumber
1	DRabid15
2	LRains21
3	TRuiz36
4	PRober45
5	KRodri49
6	BReed57
7	JRose57
8	BRodge61

Figure 38: Partial Results Set

[ ] square brackets

The square brackets allow you to define an array or range of required characters. When defining a list of possible matching values, separate them with commas or place the letters directly next to each as shown in the two samples below. Both samples return the set of accounts where the first letter is either an R or a T.

Sample list

```
SELECT AccountNumber
FROM CustomerAccount
WHERE AccountNumber LIKE '[RT]%'
;
```

	AccountNumber
1	TDudle6
2	RMondr9
3	TPittm11
4	RDavis14
5	RGaita14
6	RTalbo19
7	RMcCar25
8	RJohns28

Figure 39: Partial Results Set

```
SELECT AccountNumber
FROM CustomerAccount
WHERE AccountNumber LIKE '[R,T]%'
;
```

If you want to define a range of valid values in your where clause, use a dash between the letters representing the start and end of the range. For example, the sample below returns all records where the account number starts with the letters R, S, or T.

Sample range

```
SELECT AccountNumber
FROM CustomerAccount
WHERE AccountNumber LIKE '[R-T]%'
;
```

	AccountNumber
1	SAdams1
2	SAdams3
3	TDudle6
4	RMondr9
5	TPittm11
6	RDavis14
7	RGaita14
8	SJohns17

Figure 40: Partial Results Set

Caret (^)

The caret symbol is treated as a NOT. Use the caret to find data based on what is not there. For example, if a field is not supposed to include anything except for letters, you can search on ^A-Z as shown in the sample below.

Sample

```
SELECT C.StateProvinceCode
FROM Customer AS C
WHERE C.StateProvinceCode LIKE '%[^A-Z]%'
;
```

	StateProvinceCode
1	^M
2	^M
3	^M
4	^M
5	^M
6	^M
7	^M
8	^M

Figure 41: Partial Results Set

You can also use the combination of square brackets and the caret to locate all records that do not end in a letter or number.

 Copyright © 2018 Ann L. Weber

	I frequently get questions about why someone should use the ^ within the LIKE statement rather than simply using NOT before the LIKE. The difference is that the ^ disallows certain characters within the context of the rest of the string while NOT negates the results after the entire string analysis is completed. Although you can sometimes get the same results both ways, the caret is more flexible than using NOT before LIKE.

Escape Characters

There are two options when you need to search for characters that are reserved as wild cards for LIKE. The first method is to include the character inside of square brackets. The second method is using ESCAPE key word and define a custom escape character. This becomes necessary if one of the wild card characters is stored in your data and you need to search on it. For example, if you want to search for sale descriptions that include the phrase '30% off', you would need to "escape" the percent sign since it otherwise would be interpreted as a wildcard meaning any zero to infinite characters.

Sample square brackets

```
DECLARE @test varchar(50) = '30%'
SELECT 'Match'
WHERE @test LIKE '__[%]';
```

Sample Pipe (|) as the escape character

```
DECLARE @test varchar(50) = '30%'
SELECT 'Match'
WHERE @test LIKE '__|%' ESCAPE '|'
;
```

	Although many LIKE clauses allow the optimizer to take advantage of index seeks, leading wild cards are non-SARGable, meaning the server cannot use an index seek to locate matching rows. When possible, especially on a busy system with large data stores, avoid leading wild cards.

Try It 9 – Using LIKE

In this exercise, you will practice retrieving rows that match a pattern. You will write a set of three queries that will return the CustomerID, FirstName, and LastName fields from the Customer table for customers whose last names match the patterns requested below for each query.

1. Create a new query window and save your query to the **\Student Files** folder as **LIKE.sql**. If you need help you can refer back to the first Try It exercise in this chapter.
2. Set the database context to **RetailBankingSample**.

Copyright © 2018 Ann L. Weber

3. Write and execute a query to return the CustomerID, FirstName, and LastName fields from the Customer table for customers whose last names start with the letter A. Use the LIKE command as shown below. **11 rows** should be returned.

```
SELECT C.CustomerID, C.FirstName, C.LastName
FROM Customer AS C
WHERE LastName LIKE 'A%'
;
```

4. Modify the query to return customers with last names that start with the letters A through D. The query below includes the required WHERE clause. **73 rows** should be returned.

```
SELECT C.CustomerID, C.FirstName, C.LastName
FROM Customer AS C
WHERE LastName LIKE '[A-D]%'
;
```

5. Modify the query to return customers where the third character of the last name is a "D". **9 rows** are returned.
   Important: There are two underscores in front of the d% in the query below even though it looks like it is a single wide underscore.

```
SELECT C.CustomerID, C.FirstName, C.LastName
FROM Customer AS C
WHERE LastName LIKE '__d%'
;
```

6. Save your query and close the current SQL editor tab.

Additional SELECT Options

DISTINCT

By default, SQL returns every row in the defined result set even if it is exactly the same as another row in the result set. The DISTINCT key word tells SQL to remove any rows that are exactly the same as another row in the result set. DISTINCT applies to the whole row, and not just to the column directly after the DISTINCT keyword.

Sample

```
SELECT DISTINCT C.City, C.StateProvinceCode
FROM Customer AS C
;
```

 Copyright © 2018 Ann L. Weber

	City	StateProvinceCode
1	Adelphi	DC
2	Advance	NC
3	Alabaster	AL
4	Albany	NY
5	Albuquerque	*M
6	Allendale	SC
7	Appleton	WI
8	Arden	NC

Figure 42: Partial Results Set

As the query above displays, there are 229 distinct city and state combinations. If you run the query again without the C.StateProvinceCode column, you will find there are only 217 distinct city names. You will learn how to easily locate those city names that exist in more than one state in **Chapter 5 and Grouping Data**.

TOP

The TOP keyword should probably have been named "First" or something similar because TOP returns the first specified number of rows in the result set. As with any query, if an ORDER BY clause is not specified, the row order is determined by the order the rows are processed based on the execution plan. When an ORDER BY clause is specified, the rows are returned based on the sort order. For example, to retrieve the rows with the highest numeric values, the ORDER BY clause must specify DESC.

Syntax

```
[
    TOP (expression) [PERCENT]
    [ WITH TIES ]
]
```

Sample

```
SELECT TOP 30 AcctID, Amount
FROM AccountTransaction AS AT
ORDER BY AT.Amount DESC
;
```

The PERCENT option allows you to define the percent of total rows in the result set that will be returned. If a query without the TOP command returns 5000 rows, the same query with TOP 10 PERCENT would return 500 rows. If the number of rows in the result set is not evenly divisible for the percent defined, an additional row is returned.

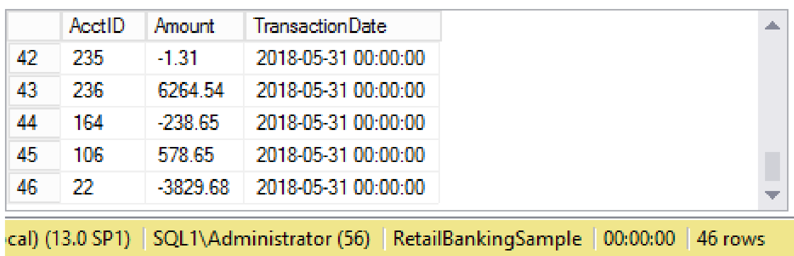
Sample

```
SELECT TOP 10 PERCENT AcctID, Amount
FROM AccountTransaction AS AT
ORDER BY AT.Amount DESC
;
```

In addition to returning the number of specified rows, TOP WITH TIES returns the TOP N records along with records that are tied with the value in the last record as defined by the number or percent in the TOP predicate. For example, the query below returns the first 30 transactions starting with those on the most recent date in the table. If there are more transactions on the same date as the 30th record, those records will also be included.

Sample

```
SELECT TOP 30 WITH TIES AcctID
   , Amount, AT.TransactionDate
FROM AccountTransaction AS AT
ORDER BY AT.TransactionDate DESC
;
```



	AcctID	Amount	TransactionDate
42	235	-1.31	2018-05-31 00:00:00
43	236	6264.54	2018-05-31 00:00:00
44	164	-238.65	2018-05-31 00:00:00
45	106	578.65	2018-05-31 00:00:00
46	22	-3829.68	2018-05-31 00:00:00

cal) (13.0 SP1) | SQL1\Administrator (56) | RetailBankingSample | 00:00:00 | 46 rows

Figure 43: Results

 Copyright © 2018 Ann L. Weber

Chapter 3 - Built-in Functions Overview

In this chapter:

How to find help on functions
Working with Functions
Mathematical Function Overview
String Function Overview
Date Time Function Overview
Nesting Functions
Understanding Data Type Conversion
Chapter 3 Lab
Answers to Exercises

Files needed:

- \Chapter 03 Functions\Inline Samples
- \Chapter 03 Functions \Try It Exercises
- \Chapter 03 Functions \Labs\

	Any starter or answer files for the Try It exercises can be found in the \Try It Exercises folder for each chapter. All code samples included within the chapter text are located in the \Inline Samples folder for each chapter.

Most programming languages include numerous functions that allow you to manipulate and work with your data. Like many other languages, T-SQL functions are typically written with the function name followed by a set of parentheses. Often, you will pass parameters to the function within the parentheses.

For example, the CONVERT function requires two parameters: the desired data type and the value that will be converted. Optionally, a third parameter will pass in a formatting style for certain data type, as shown below:

```
CONVERT(varchar(20), TransactionDate, 109)
```

How to find help on functions

Because of the large number of functions, it is not feasible to cover every function. Our goal in this class is to introduce you to some of the more common functions and to provide you with some function examples, including examples of functions nested inside of other functions.

Once you understand the basics of working with functions, you can use the Microsoft online documentation to see a list of available functions, which parameters are required and which are optional, as well as sample code. Although the URL changes from time to time, at the time of publication, the T-SQL Documentation can be found at https://docs.microsoft.com/en-us/sql/t-sql/functions/functions and is shown in Figure 44.

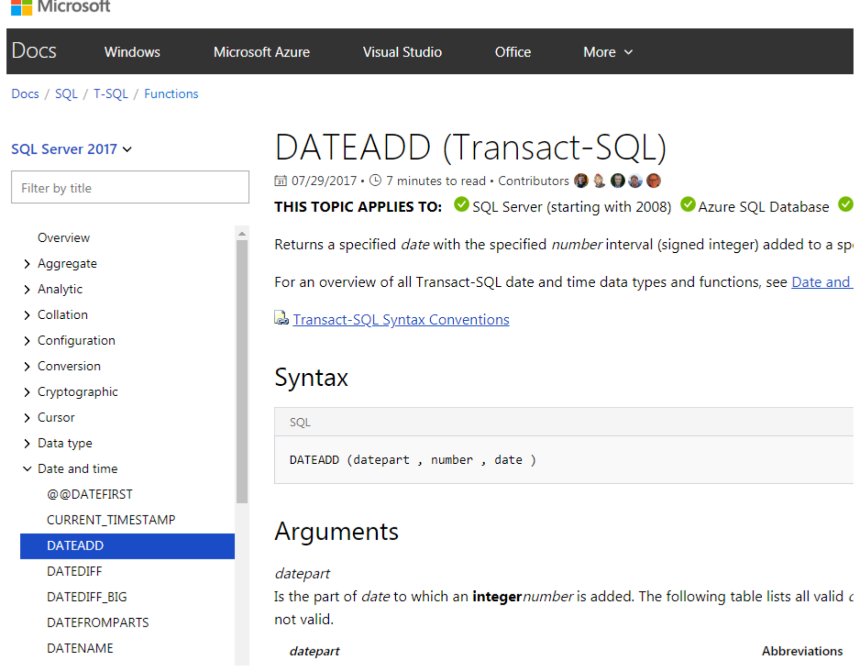


Figure 44: T-SQL Documentation

 Copyright © 2018 Ann L. Weber

The main types of functions are listed on the Overview page, along with the categories of scalar functions. The left side of the page includes expandable sections from which you can select a function to get more detailed help.

Within SQL Server Management Studio (SSMS), pressing F1 while your cursor is inside a query will attempt to pull up the help based on the word where the cursor is located or a highlighted selection.

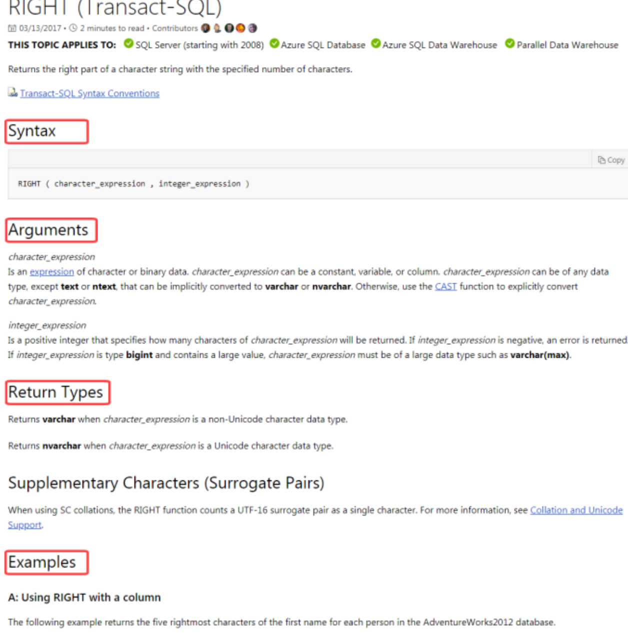
Typically the following search strings, or something similar, will work with your favorite search engine to get you the help you want:

- Microsoft docs SQL built-in functions
- Microsoft docs SQL CAST CONVERT
- Microsoft docs SQL LEN

Understanding Function Help

Each help page includes the command syntax near the top. The next section includes a definition for each argument or parameter within the syntax. A third section defines the type of data the function returns. Then, if you scroll all the way to the bottom of the help page, you will find examples of functions that utilize one of the Microsoft sample datasets such as the Adventureworks, AdventureworksDW, Northwind, or Pubs databases.

Some of the sections of the help pages are called out in Figure 45 below.



RIGHT (Transact-SQL)

03/13/2017 • 2 minutes to read • Contributors

THIS TOPIC APPLIES TO: SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Returns the right part of a character string with the specified number of characters.

Transact-SQL Syntax Conventions

Syntax

```
RIGHT ( character_expression , integer_expression )
```

Arguments

character_expression
Is an expression of character or binary data. *character_expression* can be a constant, variable, or column. *character_expression* can be of any data type, except **text** or **ntext**, that can be implicitly converted to **varchar** or **nvarchar**. Otherwise, use the CAST function to explicitly convert *character_expression*.

integer_expression
Is a positive integer that specifies how many characters of *character_expression* will be returned. If *integer_expression* is negative, an error is returned. If *integer_expression* is type **bigint** and contains a large value, *character_expression* must be of a large data type such as **varchar(max)**.

Return Types

Returns **varchar** when *character_expression* is a non-Unicode character data type.

Returns **nvarchar** when *character_expression* is a Unicode character data type.

Supplementary Characters (Surrogate Pairs)

When using SC collations, the RIGHT function counts a UTF-16 surrogate pair as a single character. For more information, see Collation and Unicode Support.

Examples

A: Using RIGHT with a column

The following example returns the five rightmost characters of the first name for each person in the AdventureWorks2012 database.

Figure 45: Help Page Sections

Try It 1 – Finding help

In this exercise, you will practice finding and interpreting the help provided by Microsoft for Built-in functions. If needed, the queries can be opened from \Chapter 03 Functions\Try It Exercises.

	This practice requires Internet connectivity.

1. If necessary, open SSMS, connect to the relational database engine, and open a new query window. Click the **Save** icon Standard toolbar and then browse to the **\Student Files** folder. Type **Finding Help.sql** in the File name box and then click **Save**.
2. Type the following code, highlight the word CONVERT, and then press F1.

```
SELECT CONVERT(varchar(30), GETDATE(), 109);
```

3. Review the help page. Notice that just below the title is a list of versions of SQL Server to which this topic applies. This page is different than the typical help page because it includes examples immediately below the applicable versions area.
4. Scroll to the bottom of the help page and notice the additional examples provided.
5. Open your favorite browser, type the following in the search bar, and then press the Enter key. Click on the first link.

```
Microsoft docs SQL built-in functions
```

6. Review the help provided. Notice that the functions are organized into sections.
7. On the left side of the page, expand the String section as shown in Figure 46, and then click the link for SUBSTRING.

 Copyright © 2018 Ann L. Weber

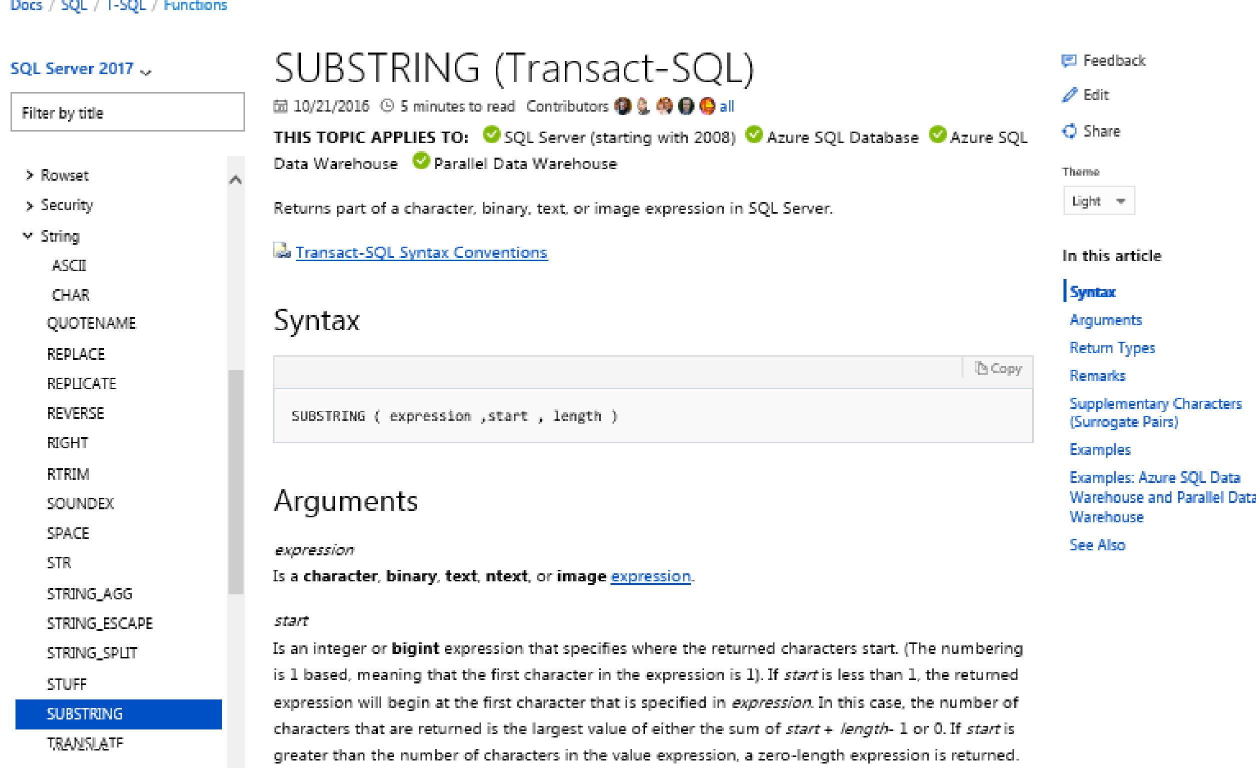


Figure 46: Substring Help

8. Locate and copy the code under Example A Using SUBSTRING with a character string and then paste it into a new query window.
9. Verify that the master database is selected in the drop-down list, and then click Execute or press F5 to run the sample. You will learn more about how the SUBSTRING function works later in this chapter.
10. Type and execute the query below to change the database context to the RetailBankingSample.

```
USE RetailBankingSample;
```

11. Modify the query to return the following three columns without a WHERE clause.
    a. FirstName – pulled directly from the Customer table
    b. Initial – the first character of the FirstName column
    c. ThirdAndFourthCharacters – the third and fourth characters from the FirstName column

```
SELECT FirstName, SUBSTRING(FirstName, 1, 1) AS
Initial
       ,  SUBSTRING(FirstName, 3, 2) AS
```

```
ThirdAndFourthCharacters
FROM Customer ;
```

12. Execute the query.
13. Save your query. Close your web browser and query window. Leave SSMS open for the next Try It exercise.

Working with Functions

The function help is grouped into categories of similar functions. We will follow this pattern for the rest of the chapter.

Most functions are referred to as scalar functions, meaning they return a single value each time they are run. Because of the number of scalar functions, these are broken down into additional categories. In this class, we will look primarily at the scalar functions in the Mathematical, String, and Date/Time categories.

In addition to the scalar functions, SQL includes several non-scalar categories of functions including:

- Aggregate Functions – although these return a single value like scalar functions, the value is derived by summarizing multiple input values. You will learn more about aggregate functions in **Chapter 5 Aggregating and Grouping Data**.
- Analytic Functions – like aggregate functions, analytic functions compute an aggregate, but they differ because analytic functions may return multiple rows per group.
- Ranking Functions – return a ranking value for each row in a partition (window). Ranking functions will be covered in **Chapter 5 Aggregating and Grouping Data**.

Mathematical Function Overview

Probably the most straightforward functions are the Mathematical functions. All standard mathematical functions (LOG, PI, SQRT, SIN, etc.) are available. To understand the mathematical functions, you must first understand the numeric data types available in SQL Server.

Numeric Data Types

The following table contains a list of the available numeric data types and some additional information about these data types.

Data Type	Value Range	Arguments	Size in Bytes
decimal / numeric	- 10^38 +1 through 10^38 - 1	decimal (p, [s]) where p(precision) is total digits and s(scale) is digits to the right of the decimal point.	5-17
float	- 1.79E+308 to -2.23E-308, 0 and 2.23E-308 to 1.79E+308	Float[(n)] where n is the number of bits used to store the mantissa of the float number in scientific notation.	4 or 8

 Copyright © 2018 Ann L. Weber

real	- 3.40E + 38 to -1.18E - 38, 0 and 1.18E - 38 to 3.40E + 38		4
int	-2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647)		4
bigint	-2^63 (-9,223,372,036,854,775,808) to 2^63-1 (9,223,372,036,854,775,807)		8
smallint	-2^15 (-32,768) to 2^15-1 (32,767)		2
tinyint	0 to 255		1
money*	-922,337,203,685,477.5808 to 922,337,203,685,477.5807		8
smallmoney*	- 214,748.3648 to 214,748.3647		4

* money and smallmoney have a scale of four decimal places. If you need to track more or fewer decimal places, you can use the numeric/decimal data type which allows you to define the scale. Even if you are using money, you can use the CONVERT command to display the value as two decimal places, but then, you might have rounding errors.

If you are working with a lot of digits to the right of the decimal place and you are doing more than simply addition and subtraction, you need to be aware of some complex and surprising rules. An example demonstrating this can be found below and in the Inline Samples 03.sql file. You can read more about this phenomena at https://docs.microsoft.com/en-us/sql/t-sql/data-types/precision-scale-and-length-transact-sql?view=sql-server-2017 and https://blogs.msdn.microsoft.com/sqlprogrammability/2006/03/29/multiplication-and-division-with-numerics/.

Sample

```
DECLARE @largenumeric1 numeric(38,16)
             = 100000000.1234567812345678
   , @numeric2 numeric (20,8) = 100.11111111

SELECT @largenumeric1 * @numeric2;
```

Mathematical Functions

A few more common mathematical functions are covered below. Each function accepts a numeric expression and returns a value with a datatype determined by the type of numeric expression that was passed in. If you want numbers to be returned with a specific precision or scale, use the CONVERT or CAST commands to display the results in a specific format, precision, and/or scale. The Try It practice is designed help you work with and compare some of these functions.

	You can read more about the parameters to be passed in and the returned value data types under each function linked at: https://docs.microsoft.com/en-us/sql/t-sql/functions/mathematical-functions-transact-sql?view=sql-server-2017.

Round	ROUND(numeric_expression , length [,function])	Either rounds or truncates the numeric expression to the defined length. The optional function argument will cause the returned value to be truncated instead of rounded unless you pass in a zero (0).
Ceiling	CEILING(numeric_expression)	Returns the smallest integer greater than or equal the numeric expression.
Floor	FLOOR(numeric_expression)	Returns the largest integer less than or equal the numeric expression.
ABS	ABS(numeric_expression)	Returns the absolute value of the numeric expression.

Try It 2 – Mathematical Functions

By performing the following steps, you will compare the differences between the round, ceiling, and floor functions. If you don't want to type the queries, you can use the Try It 2 - Mathematical Functions.sql script in the \Chapter 03 Functions\Try It Exercises folder.

1. If necessary, open SSMS and create a new query.
2. Verify that the active database is set to RetailBankingSample.
3. Type in and execute the following query to test the results of the ROUND function with different options. For the length, 0 returns whole numbers, positive numbers specify how many places to the right of the decimal point to round to, and negative numbers represent round to 10s, 100s, etc.

```
SELECT AT.AccountTransactionID, AT.Amount
     , ROUND(AT.Amount, 1, 0) AS RoundedDefault
     , ROUND(AT.Amount, 1, 1) AS Truncated
     , ROUND(AT.Amount, 0) AS RoundedDefaultToWholeNumbers
     , ROUND(AT.Amount, 0,1) AS TruncatedToWholeNumbers
     , ROUND(AT.Amount, -1) AS RoundedDefaultToTens
     , ROUND(AT.Amount, -1,1) AS TruncatedToTens
FROM AccountTransaction AS AT
;
```

4. Add two more columns to the query above to display the ceiling and floor values for each account transaction as shown in the query below. Execute the query and review the data. Pay particular attention to how negative numbers behave with ceiling and floor vs the truncate version of ROUND with whole numbers.

 Copyright © 2018 Ann L. Weber

```
SELECT AT.AccountTransactionID, AT.Amount
     , ROUND(AT.Amount, 1, 0) AS RoundedDefaultToTenths
     , ROUND(AT.Amount, 1, 1) AS TruncatedToTenths
     , ROUND(AT.Amount, 0) AS RoundedDefaultToWholeNumbers
     , ROUND(AT.Amount, 0,1) AS TruncatedToWholeNumbers
     , ROUND(AT.Amount, -1) AS RoundedDefaultToTens
     , ROUND(AT.Amount, -1,1) AS TruncatedToTens
     , CEILING(AT.Amount) AS CeilingValue
     , FLOOR(AT.Amount) AS FloorValue
FROM AccountTransaction AS AT
;
```

5. If you created a new query, click **File | Save** (or click the Save icon). Browse to the **\Student Files** folder, change the File name to **Ch3TryIt2.sql**, and then click **Save**. If you used the answer key, click **File | Save Try It 2 - Mathematical Functions.sql As** and save the file to the **\Student Files** folder.

	I once had a situation where we were dividing fundraiser money between the students and the organization. Although SQL can carry money out to more decimal places, there is still nothing smaller than a penny. I used the FLOOR function to round all values for the organization down to the nearest penny and the ceiling to always round the student up to the nearest penny so that we didn't end up with a summed value different from the actual amount coming in. In this case, rounding pennies in favor of students was approved. Other situations will require alternatives, such as using the ROUND function, to verify that rounding errors don't cause actual values to be different from the derived values.

String Function Overview

String functions are some of the most used functions in SQL. By using string functions, you can do the following and more:

- Determine the length of a string or the location of a specific pattern within a string.
- Trim spaces off the beginning, end, or both sides of a string.
- Retrieve a certain number of characters from the beginning, end, or even the middle of a string.
- Replace one value with another.
- Replicate a value a specific number of times.

When solving a problem, you will likely find more than one way to achieve the goal. If the query is only used once, ensuring that the query returns the correct data in the correct format is the only important

thing to test. But, if the query will be used over and over again, you should also test the performance of the different methods and use the option with the least amount of overhead and best performance.

String Related Data Types

The table below summarizes the current string-related data types. The text and ntext data types have been deprecated and are therefore not covered in class.

Data Type	Description	Arguments	Storage
char	Fixed length character data	char([n]) (1)	1 byte per defined character
varchar	Variable length character data	varchar([n] | max) (1,2)	1 byte per filled character (3)
nchar	Fixed Unicode character data	nchar [(n)] (4)	2 bytes per defined character
nvarchar	Variable length Unicode character data	nvarchar([n] | max) (2,4)	2 bytes per filled character (3)

(1) n is value between 1 and 8000 representing the number of characters allowed.

(2) max indicates maximum storage of 2 GB.

(3) there is a small amount of additional overhead when using variable length fields.

(4) n is value between 1 and 4000 representing the number of characters allowed.

The character fields can store the 256 characters listed in the ASCII character set, but Unicode data types use two bytes of storage per character. Thus, Unicode data types provide you with a much greater number of characters to choose from. The set of characters that can be stored in the column is defined by the collation, which is defined as a character set and sort order. SQL Server includes many different collations.

When working with character fields, you typically surround the text with single quotes ('). Be careful if you are copying and pasting from other products, such as Microsoft Word, because these tools often convert straight quotes to smart quotes. The smart quotes will cause your query to fail.

When working with Unicode data, the opening single quote is preceded by a capital N as shown in the sample below. Normally T-SQL is not case-sensitive, but you must capitalize the N in order for this statement to execute successfully.

Sample

```
DECLARE @myunicodevar nchar(50) = N'This is my value'
```

String Functions

The following table lists each string function along with a brief description. The table is sorted with similar or dependent functionalities grouped together. The more commonly used functions are near the top.

 Copyright © 2018 Ann L. Weber

The syntax, samples, and full descriptions for the string functions listed in the table below can be found at: https://docs.microsoft.com/en-us/sql/t-sql/functions/string-functions-transact-sql?view=sql-server-2017. Unlike the list below, the list at this URL is sorted in alphabetical order.

CONCAT	Concatenates any two or more strings.
CONCAT_WS	Concatenates any number of expressions with a specified delimiter. (new in SQL 2017)
FORMAT	Formats values of different data types using either pre-defined standard formats or custom formats. Locale definitions can be applied to standard formats as shown in the Try It exercise below.
LEFT	Returns the specified number of characters starting from the left side of a string.
RIGHT	Returns the specified number of characters starting from the right side of a string.
SUBSTRING	Returns the specified number of characters starting at the defined location and continuing to the right.
LEN	Returns the number of characters in a string. Trailing spaces are not counted, but leading spaces are. This is in contrast to the DATALENGTH function which is a data type function that returns the number of bytes in a string. For character data types, both leading and training spaces will count in the DATALENGTH but not the LEN command.
LOWER	Changes the case of all characters to lower case.
UPPER	Changes the case of all characters to lower case.
LTRIM	Removes all spaces from the left side of a string expression.
RTRIM	Removes all spaces from the right side of a string expression.
TRIM	(Starting with SQL 2017) Removes all spaces from both sides of a string expression.
CHARINDEX	Locates the starting position of a string within a string.
PATINDEX	Is a cross between CHARINDEX and LIKE. The pattern is defined with wild cards similar to LIKE, but returns the numeric position where the pattern starts, as in CHARINDEX.
QUOTENAME	Adds a defined quoted identifier (by default brackets are used) around a string expression. Valid identifiers are the single quote, a double quote, or square brackets.
REPLACE	Replaces all occurrences of one string character with another specified character. Both this function and the QUOTENAME function can be helpful when you need to export data with a specific delimiter when a different delimiter already exists in the data.
TRANSLATE	(Starting with SQL 2017) Used as a more concise option than REPLACE when replacing multiple characters in the same string or in conjunction with geospatial coordinates.
REPLICATE	Repeats a string a specified number of times.
REVERSE	Reverses the order of the characters in a string.
ASCII	Converts the leftmost character of any string to the ASCII code for that character.
CHAR	Converts an ASCII code integer value to the character represented.
NCHAR	Converts a Unicode code integer value to the Unicode character represented.
UNICODE	Converts the leftmost character of any string to the Unicode integer value for that character.
SPACE	Returns a string of spaces of a defined length. For example, SPACE(3) is easier to interpret when reading the code than three spaces between single quotes.
STR	Converts approximate numeric data to character data of a defined length which includes the sign, decimal point, digits, and spaces. The decimal portion defines the number of places to the right of the decimal place.
STUFF	Deletes the data at the defined point and length in one string and replaces it with another string.

SOUNDEX	Returns a four-character code based on how the string sounds when spoken to evaluate the similarity of two strings.
DIFFERENCE	Returns the numeric difference between the SOUNDEX values of two string expressions.

<table>
<tr>
<td>
</td>
<td>The .NET framework format strings can be found in the following locations:
• Standard Numeric Format Strings - https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings
• Custom Numeric Format Strings – https://docs.microsoft.com/en-us/dotnet/standard/base-types/custom-numeric-format-strings
• Standard Date and Time Format Strings – https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-date-and-time-format-strings
• Custom Date and Time Format Strings – https://docs.microsoft.com/en-us/dotnet/standard/base-types/custom-date-and-time-format-strings</td>
</tr>
</table>

Try It 3 – String Functions

In this exercise, you will practice working with a number of the string functions. Throughout the course, you may be asked to use other functions that you haven't yet had the opportunity to use. Remember to use F1 from SSMS or your favorite search engine to find information on the parameters needed, other syntax help, and samples.

1. In SSMS, click **File** | **Open** | **File** (or click the Open file icon) and browse to open **\Chapter 03 Functions\Try It 3 – String Functions Starter.sql**. Click File | **Save Try It 3 – String Functions Starter.sql As** to open the Save As dialog box, browse to the **\Student Files** folder, and save the query to the new location.
2. Review, and execute all of the commands under Step #2. Why are there extra spaces in some locations and not others?
3. Review, and execute all commands under Step #3.
4. Review and discuss the results.
5. Click **File** | **Save Try It 3 – String Functions Starter.sql As**, and then browse to the **\Student Files** folder. Click **Save** to save the file in the new location.
6. Below the existing queries, write a query that will **trim** the trailing characters off the **FirstName** field from the **Customer** table and then concatenate this value with a single space and the **LastName** field from the same table. Alias this

 Copyright © 2018 Ann L. Weber

column as FullName.
If you need help, the answer is provided in the **\Chapter 03 Functions\Try It 3 – String Functions Answer.sql** file.
Note: If you have SQL Server 2017 or later, the TRIM function will trim extra spaces off both sides of a character field at the same time. In prior versions of SQL Server, you would have to nest an LTRIM and RTRIM inside of each other.

7. There are times when you will want to take dates and format them in a specific way immediately, rather than using a reporting tool to format the results. In the next step, you will write a query to format the Amount from the Transaction Account table with both the US English and France French locale settings. The query should include the AccountTransactionID, AcctID, TransactionDate and TransactionType columns in addition to the formatted amount. A partial result set is shown in Figure 47.

	AccountTransactionID	AcctID	TransactionDate	TransactionType	USMoneyAmount	FrenchMoneyAmount
1	1	12	2014-10-14 00:00:00	Mobile App	$2,978.15	2 978,15 €
2	2	12	2018-05-28 00:00:00	ATM	($1,135.97)	-1 135,97 €
3	3	12	2017-05-07 00:00:00	Direct Deposit	$3,991.69	3 991,69 €
4	4	12	2016-09-26 00:00:00	ATM	($2,817.13)	-2 817,13 €
5	5	12	2014-03-12 00:00:00	Direct Deposit	$2,357.13	2 357,13 €
6	6	12	2016-06-01 00:00:00	ATM	$1,043.05	1 043,05 €
7	7	12	2018-04-02 00:00:00	Mobile App	($1,830.80)	-1 830,80 €
8	8	12	2015-10-16 00:00:00	Debit Card	($1,542.84)	-1 542,84 €

Figure 47: Partial Results Set

8. Review, type and then execute the following query to format the result set as US money and French money.

```
SELECT AT.AccountTransactionID, AT.AcctID
  , AT.TransactionDate, AT.TransactionType
  , FORMAT(AT.Amount, 'C2', 'en-US') AS USMoneyAmount
  , FORMAT(AT.Amount, 'C2', 'fr-FR') AS FrenchMoneyAmount
FROM AccountTransaction AS AT
;
```

9. Modify the existing query to use the FORMAT function and the standard small date format code (‘d’) to format the transaction date column in the above query to both a US English and France French locale settings. Both columns should be aliased with names representing the locale formats.
If you need the help, the answer is provided in the **\Chapter 03 Functions\Try It 3 – String Functions Answer.sql** file under **Step #9**. The results should look similar to Figure 48.

	AccountTransactionID	AcctID	TransactionType	USMoneyAmount	FrenchMoneyAmount	USDate	FrenchDate
1	1	12	Mobile App	$2,978.15	2 978,15 €	10/14/2014	14/10/2014
2	2	12	ATM	($1,135.97)	-1 135,97 €	5/28/2018	28/05/2018
3	3	12	Direct Deposit	$3,991.69	3 991,69 €	5/7/2017	07/05/2017
4	4	12	ATM	($2,817.13)	-2 817,13 €	9/26/2016	26/09/2016
5	5	12	Direct Deposit	$2,357.13	2 357,13 €	3/12/2014	12/03/2014
6	6	12	ATM	$1,043.05	1 043,05 €	6/1/2016	01/06/2016
7	7	12	Mobile App	($1,830.80)	-1 830,80 €	4/2/2018	02/04/2018
8	8	12	Debit Card	($1,542.84)	-1 542,84 €	10/16/2015	16/10/2015

Figure 48: Partial Results Set

10. Add one more column that displays the transaction date in the format Monday 7 May 2018. Use the help for .NET custom date formats at https://docs.microsoft.com/en-us/dotnet/standard/base-types/custom-date-and-time-format-strings for the required string. If you need help, the answer is provided in the **\Chapter 03 Functions\Try It 3 – String Functions Answer.sql** file under **Step #10**. The results should look similar to Figure 49.

    Hints: The entire custom string should be enclosed in single quotes, not double quotes. Also, remember that the formatting characters are case sensitive. Lower case m is minutes while upper case M is months.

	AccountTransactionID	AcctID	TransactionType	USMoneyAmount	FrenchMoneyAmount	USDate	FrenchDate	CustomDate
1	1	12	Mobile App	$2,978.15	2 978,15 €	10/14/2014	14/10/2014	Tuesday 14 October 2014
2	2	12	ATM	($1,135.97)	-1 135,97 €	5/28/2018	28/05/2018	Monday 28 May 2018
3	3	12	Direct Deposit	$3,991.69	3 991,69 €	5/7/2017	07/05/2017	Sunday 07 May 2017
4	4	12	ATM	($2,817.13)	-2 817,13 €	9/26/2016	26/09/2016	Monday 26 September 2016
5	5	12	Direct Deposit	$2,357.13	2 357,13 €	3/12/2014	12/03/2014	Wednesday 12 March 2014
6	6	12	ATM	$1,043.05	1 043,05 €	6/1/2016	01/06/2016	Wednesday 01 June 2016
7	7	12	Mobile App	($1,830.80)	-1 830,80 €	4/2/2018	02/04/2018	Monday 02 April 2018
8	8	12	Debit Card	($1,542.84)	-1 542,84 €	10/16/2015	16/10/2015	Friday 16 October 2015

Figure 49: Partial Results Set

11. Save and close your Try It 3 – String Function Starter.sql file. Leave your SSMS open for the next Try It exercise.

Date Time Function Overview

Another set of popular scalar functions are those that work with dates and time.

Understanding Date and Time Data Types

Before we can start working with the date and time functions, we first need to understand the data and time data types that are available in SQL Server.

Data Type	Accuracy	Date Range	Size in Bytes
date[(1)]	To the day	January 1, 0001 to December 31, 9999	3
time[(1)]	Up to 100ns	n/a	5

 Copyright © 2018 Ann L. Weber

datetime	Rounded to .000, .003, or .007[2]	January 1, 1753 to December 31, 9999	8
datetime2[1]	From whole second to 100ns	January 1, 0001 to December 31, 9999	6-8 depending on precision.
smalldatetime[1]	One minute[1]	January 1, 1900 to June 6, 2079	4
Datetimeoffset[1]	From whole second to 100ns	January 1, 0001 to December 31, 9999	10[3]

[1] Data types available in SQL 2008 and later only.

[2] Datetime rounds .998 down to .997 while .998 rounds up to the next whole second. \Chapter 03 Functions\Inline Samples\DateTimeRounding.sql has a sample of what happens with this rounding on December 31st at 11:59:59 pm.

[3]Includes offset from UTC and a default accuracy to 100ns. Size does not vary based on accuracy.

The precision of both the time and datetime2 data types can be configured, allowing you to include 0 to 7 decimal places representing fractional seconds.

Sample

```
DECLARE @secondsvar datetime2(0) = GETDATE()
   , @nanosecondvar datetime2(7) = GETDATE()

SELECT @secondsvar AS DisplayWholeSeconds
   , @nanosecondvar AS Display100ns
;
```

Result Set

	DisplayWholeSeconds	Display100ns
1	2018-05-14 17:30:15	2018-05-14 17:30:15.3000000

Although the majority of production databases still use the datetime data type, efficiencies in both performance and functionality can be gained by carefully picking the correct data type for each situation.

Date Retrieval

All of the date retrieval functions except for CURRENT_TIMESTAMP have the same syntax - the function name followed by open and close parentheses, as in the samples below. Additionally, all date functions are returning the current date, time, and, optionally, the time zone offset from the server. The date and time returned will be based on the server settings, not the local computer.

The differences come from the data types that are returned. Because the data types define the precision and range of the date information returned, choose the function you use based on the information that you need.

Syntax

```
GETDATE ( )
```

```
CURRENT_TIMESTAMP
```

Sample

```
SELECT GETDATE() AS CurrentDate1
  , CURRENT_TIMESTAMP as CurrentDate2
  , SYSDATETIME() AS CurrentDateAsDatetime2
;
```

Result Set

	CurrentDate1	CurrentDate2	CurrentDateAsDatetime2
1	2018-05-14 17:37:15.423	2018-05-14 17:37:15.423	2018-05-14 17:37:15.4264185

GETDATE, GETUTCDATE, CURRENT_TIMESTAMP

GETUTCDATE is available only in SQL 2008 and later

This first set of date functions return results using the datetime data type.

The GETDATE and CURRENT_TIMESTAMP functions return identical results. CURRENT_TIMESTAMP is the ANSI equivalent, but isn't very prevalent in Microsoft SQL code.

GETUTCDATE returns the current Coordinated Universal Time (UTC) time based off the SQL Server's current date, time and offset. UTC does not observe daylight savings time. For example, if my server is located in the US in Eastern Daylight Savings time and the current time is exactly 11:00 am on May 2, 2018, GETUTCDATE will return 2018-05-02 15:00:00.000. If this same command were run on a server defined to NOT observe daylight savings time, the server would report 2018-05-02 16:00:00.000.

SYSDATETIME, SYSUTCDATE (*available only in SQL 2008 and later*)

Like GETDATE, SYSDATETIME retrieves the current date and time from the server, while SYSUCTDATE retrieves the current UTC date and time based on the SQL Server's time and configuration. The difference from the last set of functions is that SYSDATETIME and SYSUTCDATE return the data by using the datetime2 data type.

Although there are the newer Date and Time stand-alone data types, there are currently no functions to retrieve just the Date, or just the Time. You will need to first retrieve the date and time, and then the server will implicitly convert to either the Date or the Time to match the data type where the data is being placed. To avoid rounding confusion, use SYSDATETIME rather than GETDATE. This is because the SYSDATETIME function and the Datetime2 and Time data types all use the same accuracy, thus avoiding rounding errors or confusion.

Sample

```
DECLARE @mydatetime datetime = '20171231 23:59:59.999'
   , @mydatetime2 datetime2 = '20171231 23:59:59.999'
   , @mydatefromdatetime date
   , @mydatefromdatetime2 date

SET @mydatefromdatetime = @mydatetime
SET @mydatefromdatetime2 = @mydatetime2;
```


Copyright © 2018 Ann L. Weber

```
SELECT @mydatefromdatetime AS DatetimeSample
   , @mydatefromdatetime2 AS DateTime2Sample
;
```

Result Set

	DatetimeSample	DateTime2Sample
1	2018-01-01	2017-12-31

Although this sample uses hard coded values to avoid trying to capture data at exactly the correct millisecond, you can clearly see how the implicit conversion that occurs with the datetime data type rounds to the next second. This could cause issues, for example, on December 31st, just before midnight, because SQL Server would round the Date up to the next year.

Whenever possible, data type conversions, either implicit or explicit, should be avoided. If you are placing the current date and time into a column with the datetime data type, then use GETDATE(). If the column is defined as datetime2, then use SYSDATETIME().

SYSDATETIMEOFFSET (*available only in SQL 2008 and later*)

The SYSDATETIMEOFFSET function retrieves not only the current date and time from the SQL Server, it also retrieves the offset from UTC and includes this when placing the data into the result with a datatimeoffset data type.

Syntax

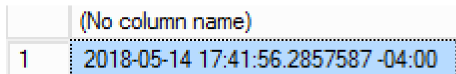
```
SYSDATETIMEOFFSET ( )
```

Sample

```
DECLARE @testsysdatetimeoffset datetimeoffset
                = SYSDATETIMEOFFSET();

SELECT @testsysdatetimeoffset;
```

Result Set



	(No column name)
1	2018-05-14 17:41:56.2857587 -04:00

What many people don't understand is that when you use one of these date retrieval functions that does not maintain offset information to add information to your table, you are taking a snapshot of the current server date and time, and you will not know what time zone the server was in, what the server settings were, etc. I had an organization that I consulted for that put all their servers in Standard time while all the humans and client

	computers worked in daylight savings time. It took a long time to explain to them that in the summer, when someone put in a work order at 3pm (DST) on their computers, it was only 2pm (standard time) on the server, so 2pm was being entered in the field. They were using datetime as their data type, so the offset was not being noted. Thus, people were confused by the times on their work orders.

Manipulating Dates

DATEADD

DATEADD allows you to add or subtract any portion of a date or time to an existing value with the date and/or time included.

The result set data type is determined by the data being passed into the function. For example, if a variable with the datetime data type is passed in, the results are also specified as datetime. If datetime2 data is passed in, the results are datetime2. This can be tested by counting the number of decimal places in the result or testing out of bound ranges by subtracting enough years to go beyond the datetime limit of 1753.

Syntax

```
DATEADD (datepart, number, date)
```

Sample

```
SELECT DATEADD(mm, -1, GETDATE())AS OneMonthAgo
   , DATEADD(mi, 30, GETDATE()) AS [30MinutesFromNow]
;
```

Result Set

	OneMonthAgo	30MinutesFromNow
1	2018-04-14 17:44:15.503	2018-05-14 18:14:15.503

As shown in the syntax and sample above, the first parameter is the portion of the date you are using for the calculation. For example, **yy** is year, **mm** is month, **dd** is day, **dw** is day of the week, **mi** is minute. Note: In this function, the date part abbreviations are not case sensitive.

The second parameter is a number representing how far into the future (positive numbers) or past (negative numbers) that you want to use for your calculation.

The third and final parameter is the date expression. This can be the data returned from a function such as DATEFROMSTRING, GETDATE or SYSDATETIME, a column from a table, or a string that represents a date.

The full explanation of this command and the options for the datepart parameter can be found at https://docs.microsoft.com/en-us/sql/t-sql/functions/dateadd-transact-sql?view=sql-server-2017.

 Copyright © 2018 Ann L. Weber

DATEDIFF and DATEDIFF_BIG

DATEDIFF_BIG available only in SQL 2016 and later

DATEDIFF and DATADIFF_BIG allow you to compare two dates and determine the length of time between them. Like many of the other date functions, you define the "datepart" that you want to work with using the abbreviations documented in the online help.

The important difference between DATEADD and DATEDIFF is that DATEADD is only working with one date and then adding a defined number of periods into the future or past. DATEDIFF is comparing two dates in an interval, such as days, weeks, or hours.

Remember - the order of the dates within the query matters. One way to test this is to simply pass the two date fields into the function. If the function returns negative numbers when you expect positive numbers, switch the order of the fields. Try thinking of it as a subtraction word problem rather than how you write it in math.

For example, if you want to do the following:

Today's Date – Transaction Date

You would phrase my math sentence as:

Subtract the Transaction Date from Today's Date

Likewise, the order would be similar to the Sample below where TransactionDate comes before GETDATE() within the DATEDIFF function.

Syntax

```
DATEDIFF ( datepart , startdate , enddate )
```

Along the same theme as above,

Sample

```
SELECT AT.AcctID, AT.TransactionDate
   , DATEDIFF(dd, TransactionDate, GETDATE())
        AS NumberofDaysAgo
FROM AccountTransaction AS AT
;
```

Result Set

	AcctID	TransactionDate	NumberofDaysAgo
1	12	2014-10-14 00:00:00	1308
2	12	2018-05-28 00:00:00	-14
3	12	2017-05-07 00:00:00	372
4	12	2016-09-26 00:00:00	595
5	12	2014-03-12 00:00:00	1524
6	12	2016-06-01 00:00:00	712

Copyright © 2018 Ann L. Weber

<table>
<tr>
<td>
</td>
<td>If you need to round the datetime field to the nearest hour, either on writes or reads, use this little trick. The code is simple to understand, but the performance could take a little hit. Because the article was written before 2008, it talks about rounding to the day portion. This is no longer necessary. If you want to round to the day with no time, simply convert the value to the date datatype.

You can find an explanation and code samples at http://improve.dk/sql-server-datetime-rounding-made-easy/</td>
</tr>
</table>

Try It 4 – DATEADD and DATEDIFF

In the following exercise you will write a query to return the date 30 years from the opening date for all 30-year mortgage accounts (AccountTypeIDs 6 and 9). If there are transactions with future dates, these dates should appear as negative numbers. Historical dates should appear as positive numbers. You will then return the number of days since the most recent transaction for these same account types.

A partial result set from the first query can be seen in Figure 50.

	AccountID	OpeningDate
1	1	2007-03-29
2	16	2011-11-22
3	18	2010-02-05
4	27	1988-04-23
5	36	2017-01-04
6	59	2012-06-28
7	65	1983-03-30
8	67	2007-04-12

Figure 50: Partial Results Set

1. Open a new query window.
2. Write and execute a query to return the AcctID and OpeningDate fields from the LoanTransaction table aliased as LT. Limit the result set to include accounts with an AccountTypeID of either 6 or 9 as shown in the query below. **47 rows** should be returned.

```
SELECT A.AccountID
FROM Account AS A
WHERE AccountTypeID IN (6,9)
;
```

3. Add another column aliased to LoanCompletion that adds 30 years to the OpeningDate field as shown below and then execute the query. 47 rows should be returned.

```
SELECT A.AccountID, A.OpeningDate
```

 Copyright © 2018 Ann L. Weber

```
     , DATEADD(YY, 30, A.OpeningDate) AS LoanCompletion
FROM Account AS A
WHERE AccountTypeID IN (6,9)
;
```

4. Below your first query, write and execute a query that returns how many days it has been since the most recent transaction in the LoanTransaction table for the accounts with AccountTypeIDs of 6 or 9. You'll need to compare the current date and the TransactionDate. Use the MAX aggregate function to retrieve the most recent order date. You will learn more about aggregate functions in **Chapter 5 Aggregating and Grouping Data**. A sample query and a partial result set are included below. **47 rows** will be returned.

```
SELECT LT.AcctID
     , DATEDIFF(dd, MAX(TransactionDate), getdate()) AS
DaysSinceLastTransaction
FROM LoanTransaction AS LT
     INNER JOIN Account AS A
     ON LT.AcctID = A.AccountID
WHERE AccountTypeID IN (6,9)
GROUP BY AcctID
ORDER BY AcctID
;
```

	AcctID	DaysSinceLastTransaction
1	1	-16
2	16	-22
3	18	7
4	27	15
5	36	9
6	59	-22
7	65	1858
8	67	-12

Figure 51: Partial Results Set

5. Click **File** | **Save** (or click the Save icon). Browse to the **\Student Files** folder, change the File name to **Try It 4.sql**, and then click **Save**.

Retrieving Parts of Dates

DATEPART and DATENAME perform very similar calculations but return different data types. DATEPART always returns the numeric representation of the date part that is requested, and DATENAME returns the character representation of the date part that is requested. For example, DATENAME returns May while DATEPART returns 5. For the day portion, the results "look" the same because, for May 10th the result is 10, but the data types of the result are different. In this case, to avoid conversions, use the function that returns the data type that you need.

Copyright © 2018 Ann L. Weber

<table>
<tr>
<td>
</td>
<td>The dw (day of week or weekday) option of DATEPART returns a numeric representation of the current day of the week. What day is considered the first day of the week is determined primarily by the locale setting for the SQL Server, but can be modified using the SET DATEFIRST command. The @@DATEFIRST function will return the current DATEFIRST setting. For more information on these commands and some special cases and rules for SET DATEFIRST see https://docs.microsoft.com/en-us/sql/t-sql/statements/set-datefirst-transact-sql?view=sql-server-2017 and https://docs.microsoft.com/en-us/sql/t-sql/functions/datefirst-transact-sql?view=sql-server-2017.</td>
</tr>
</table>

Syntax

```
DATEPART ( datepart , date )

DATENAME ( datepart , date )
```

Sample

```
SELECT DATEPART(mm, AT.TransactionDate)
FROM AccountTransaction AS AT
;
```

Result set

	(No column name)
1	10
2	5
3	5
4	9
5	3
6	6
7	4
8	10

Sample

```
SELECT DATENAME(mm, AT.TransactionDate)
FROM AccountTransaction AS AT
;
```

 Copyright © 2018 Ann L. Weber

Result set

	(No column name)
1	October
2	May
3	May
4	September
5	March
6	June
7	April
8	October

The DAY, MONTH and YEAR functions are simply shorthand used to return the numeric day, month or year from a date expression.

Syntax

```
MONTH( );
```

Samples

```
SELECT MONTH('20180204') AS MonthNumber;
```

Result Set

	MonthNumber
1	2

Try It 5 – Retrieving Date Parts

In this Try It you will practice using the DATEPART and DATENAME functions to return a result set from the Account table. All date related columns should be based on the OpeningDate column as follows:

- AccountID
- OpeningDate
- OpeningYear – 4 digit year
- OpeningMonth – Full month name spelled out
- OpeningDayofMonth – numeric representation
- OpeningDayofWeek – day of week spelled out

The result set should look similar to Figure 52.

	AccountID	OpeningDate	OpeningYear	OpeningMonth	OpeningDayofMonth	OpeningDayofWeek
1	1	2007-03-29	2007	March	29	Thursday
2	2	2009-06-13	2009	June	13	Saturday
3	3	2006-04-13	2006	April	13	Thursday
4	4	2015-10-25	2015	October	25	Sunday
5	5	2010-06-19	2010	June	19	Saturday
6	6	1985-03-31	1985	March	31	Sunday
7	7	1988-02-02	1988	February	2	Tuesday
8	8	2006-07-15	2006	July	15	Saturday

Figure 52: Partial Results Set

1. Create a new query window.
2. Click **File | Save** (or click the Save icon). Browse to the **\Student Files** folder, change the File name to **Try It 5.sql**, and then click **Save**.
3. Use the DATEPART and DATENAME functions to create the result set defined above. There are multiple correct answers to this query. Some of the options are included in the **\Chapter 03 Functions\Try It Exercises\Try It 5 - Retrieving Date Parts.sql** file.

Additional Date Functions

From Parts Functions

DATEFROMPARTS, DATETIME2FROMPARTS, DATETIMEFROMPARTS, DATETIMEOFFSETFROMPARTS, SMALLDATETIMEFROMPARTS, and TIMEFROMPARTS all allow you to create a date when the information for the different parts of the date is being retrieved from different columns or expressions. Like the date retrieval functions, the primary difference between each of these functions is which data type is returned and how many input parameters are used.

The syntax and sample below are for the DATETIMEOFFSETFROMPARTS, but the other functions all work the same way. However, the other functions don't have as many parameters.

Syntax

```
DATETIME2FROMPARTS ( year, month, day, hour
, minute, seconds, fractions, precision )
```

Sample

```
SELECT DATETIMEOFFSETFROMPARTS
         (
          2010, 12     , 31, 14     , 23     , 30
          --Year , Month, Day, Hour, Minute, Second
           , 0         , 12             , 0     , 7
          --fractions, offset hr, offset minute, precision
         ) AS Result;
```

Result Set

	Result
1	2010-12-31 14:23:30.0000000 +12:00

 Copyright © 2018 Ann L. Weber

TODATETIMEOFFSET

The TODATETIMEOFFSET function accepts two parameters - a datetime2 expression and an offset/time zone definition. This function returns the combined information as a datetimeoffset data type with the fractional precision of the original datetime2 expression.

Syntax

```
TODATETIMEOFFSET ( expression , time_zone )
```

Sample

```
DECLARE @SampleDate datetime2 = '20180101 10:15:30'
SELECT @SampleDate AS Original
            , TODATETIMEOFFSET (@SampleDate, '+05:00')
                AS OffsetDateTime
;
```

Result Set

	Original	OffsetDateTime
1	2018-01-01 10:15:30	2018-01-01 10:15:30 +05:00

The time_zone parameter represents the time zone offset in minutes if an integer is used, or hours and minutes if a string is used. For example '+13:00' would be interpreted as 13 hours, while 120 would be interpreted as 2 hours. The range is +14 to -14 (in hours). The expression is interpreted in local time for the specified time_zone.

SWITCHOFFSET

The SWITCHOFFSET function allows you to modify the stored offset. You can use this function either to display the offset as a different value in the result set or in conjunction with an UPDATE statement to modify the offset in an existing row that contains a datetimeoffset column.

For example, the SQL Server is on the East Coast and data is stored with an offset of +5, but you are sending your report to users on the West Coast. To display the time relative to where the users are located, use this code:

Syntax

```
SWITCHOFFSET ( DATETIMEOFFSET, time_zone )
```

Sample

```
DECLARE @datevar datetimeoffset = '20180101 10:15:30 +5:00'

SELECT @datevar AS Original
   , SWITCHOFFSET(@datevar, '+08:00') AS NewTime
;
```

Result Set

	Original	NewTime
1	2018-01-01 10:15:30.0000000 +05:00	2018-01-01 13:15:30.0000000 +08:00

Copyright © 2018 Ann L. Weber

	You will learn more about updating data in **Chapter 9 Data Manipulation Language**.

EOMONTH

Before 2012, SQL coders had to write a CASE statement to determine the last day of any given month, including leap year rules. Since SQL 2012, we have a function named EOMONTH that does the work for us. Simply pass a date expression to EOMONTH and it will return the date of the last day of that month using the date data type.

Syntax

```
EOMONTH(date_expression)
```

Sample

```
SELECT EOMONTH('20120215') AS LeapYearEndofFebruary
    , EOMONTH('20150215') AS NonLeapYearEndofFebruary
;
```

Result Set

	LeapYearEndofFebruary	NonLeapYearEndofFebruary
1	2012-02-29	2015-02-28

ISDATE

The ISDATE function, like its counterparts for other datatypes, returns a 1 if the expression being passed in is a date and 0 if not.

Syntax

```
ISDATE ( expression )
```

Sample

```
SELECT ISDATE('20180228') AS valid
    , ISDATE('20180231') AS NoFeb31
    , ISDATE('30/12/2018') AS NotSupportedAsWritten
    , ISDATE('12/31/2018') AS SupportedOrder
;
```

Result Set

	valid	NoFeb31	NotSupportedAsWritten	SupportedOrder
1	1	0	0	1

SET DATEFORMAT

Although not a date function, if you are working with string date representations that are in a different order from the default order, use the SET DATEFORMAT command. For example, the default format for English is MDY.

 Copyright © 2018 Ann L. Weber

Syntax

```
SET DATEFORMAT { format | @format_var }
```

Sample

```
SET DATEFORMAT DMY;
```

Within a session, once this setting has been changed, it will remain in effect until the session is closed or the command is run again to return the behavior to the default settings.

	You can find additional information on SET DATEFORMAT at https://docs.microsoft.com/en-us/sql/t-sql/statements/set-dateformat-transact-sql?view=sql-server-2017. You can find a list of default settings for different languages by executing sp_helplanguage.

Try it 6 – ISDATE and EOMONTH

This Try It exercise will help you to better understand the functionality of the ISDATE and EOMONTH functions.

1. Open the Try it 6 – ISDATE and EOMONTH Starter.sql file.
2. Click **File | Save Try it 6 – ISDATE and EOMONTH Starter.sql As**, and then browse to the **\Student Files** folder. Click **Save** to save the file in the new location.
3. Highlight and execute the SELECT statement under Step #2 and review the results.
4. Under Step #3, type a command to change the date format to dmy. If you need help, you can find the code in Try it 6 – ISDATE and EOMONTH Answer.sql
5. Rerun the query and step 2 and notice the column titles no longer reflect the dates that were interpreted as valid dates in the last two columns.
6. Set the date format back to month day year.
7. Execute the statements under Step #7 and review the output of the EndofMonth column.
8. Save your query and close the current query tab. Leave SSMS open for the next Try It exercise.

Nesting Functions

To achieve the required results, you will frequently need to nest functions inside of each other. In most cases, the functions are analyzed from the inner most function to the outer function.

When working with more complex functions, writing and testing one layer at a time can be very helpful. For example, the following select statement returns the next to last portion of an email address which

typically maps to the company or organization name. Since not all names are the same length (even the top level domain can be two characters as .US is, three characters like .com, or even five as .local), we need to use the CHARINDEX function to find the at (@) sign as our starting point and the last period (.) as our ending point. Although this can be achieved in many ways, the \Chapter 03 Functions\Inline Samples\Inline Samples 03.sql file shows not only the final result, but the steps that have been taken to test each step to build the final select statement.

Sample

```
SELECT U.UserID
   , U.EmailAddress
   , Substring(U.EmailAddress
                 ,(CHARINDEX('@', U.EmailAddress) + 1)
                 ,(LEN(U.EmailAddress)
                    - CHARINDEX('.', REVERSE(U.EmailAddress))
                      - (CHARINDEX('@', U.EmailAddress))
                  )
                ) AS ExtractedCompany
FROM [User] AS U
;
```

The Phishing database sample includes email addresses for the users, but unfortunately, to avoid using actual email addresses, everyone has a domain of company.com. Every row will return the same value in the above example.

Try It 7 - Nesting Functions

In this Try It exercise, you will nest functions to create a two-character column that includes the numeric day portion of the OpeningDate from the account table. If the day is the 1st through the 9th of the month, the result should include a leading 0. A partial result set is shown in Figure 53.

	AccountID	OpeningDate	DayChar
1	1	2007-03-29	29
2	2	2009-06-13	13
3	3	2006-04-13	13
4	4	2015-10-25	25
5	5	2010-06-19	19
6	6	1985-03-31	31
7	7	1988-02-02	02
8	8	2006-07-15	15

Figure 53: Partial Results Set

1. Open a new query window and save your script to the \Student Files folder. Save the query as NestingFunctions.sql.

 Copyright © 2018 Ann L. Weber

2. Write and execute a script that will return the AccountID and OpeningDate fields from the Account table as shown below:

```
SELECT A.AccountID, A.OpeningDate
FROM Account AS A
;
```

3. Add an additional column aliased as DayChar that returns the day of the month portion of the OpeningDate field as a character as shown below:

```
SELECT A.AccountID, A.OpeningDate
   , DATENAME(dd,A.OpeningDate) AS DayChar
FROM Account AS A
;
```

4. Modify the new column to concatenate a string literal of 0 to the left side of the day information you retrieved in the prior step as follows:

```
SELECT A.AccountID, A.OpeningDate
     , '0' + DATENAME(dd,A.OpeningDate) AS DayChar
FROM Account AS A
;
```

5. Use the right function to only return the two rightmost characters to provide a 2-digit day value with leading 0's only where single digits exist. Your query should look similar to the following query and your results should look similar to Figure 53 above.

```
SELECT A.AccountID, A.OpeningDate
  , RIGHT('0' + DATENAME(dd,A.OpeningDate), 2) AS DayChar
FROM Account AS A
;
```

6. Save and close the active query window. Leave SSMS open for the next Try It exercise.

Understanding Data Type Conversion

SQL Server includes several functions for converting data. The most common of these are CAST and CONVERT. For float or real data types, you can achieve a greater level of formatting control by using the STR function to convert to character based data types. PARSE is used when converting string data into either date/time or number data types.

Although CAST and CONVERT can be used interchangeably based on your preferences, this class will focus on CONVERT to demonstrate some of the additional features available only with CONVERT. CAST, on the other hand, is ISO-compliant and can be ported more easily to other systems.

SQL Server includes implicit rules to convert data from one data type to another for comparisons and other operations. When the server cannot implicitly convert the data, you must explicitly convert it using either the CAST or CONVERT statement.

Precedence of the data type is used to determine how data is converted when two different data types are used together either in an expression or a comparison. Data types with a lower precedence are converted to the higher precedence data type. For example, from highest precedence to lowest, the integer data type is number 16 while character is number 28. Since Char is the lower precedence, the resulting expression will have an integer data type. The following Try It exercise explores these implicit conversions.

You can find additional information on data type conversion behaviors and allowed conversions at: https://docs.microsoft.com/en-us/sql/t-sql/data-types/data-type-conversion-database-engine . Additional information on data type precedence can be found at https://docs.microsoft.com/en-us/sql/t-sql/data-types/data-type-precedence-transact-sql .

Try It 8 – Exploring Implicit Conversions

In this exercise, you will write and execute a series of short queries to understand how implicit conversions work. Before executing each query, think about what you expect the result to be. If your expectations don't match the results, determine why this happened. If needed, the queries can be opened from **\Chapter 03 Functions\Try It Exercises\Try It 8 - Implicit conversion.sql**.

1. In a new query window, type each of the following queries one by one, and then execute each query. Why did you received the result that you did?

```
SELECT 'a' + 'b';

SELECT 1 + 2;

SELECT '1' + '2';

SELECT 1 + '2';

SELECT '1' + 'a';

SELECT 1 + 'a';
```

2. Save the query as ImplicitConversions.sql in the \Student Files folder.

 Copyright © 2018 Ann L. Weber

3. Close the query tab, but leave SSMS open.

The CONVERT function

The convert function requires two parameters. A third optional parameter specifies a style.

Syntax

```
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

Sample

```
SELECT CONVERT (varchar(20), A.OpeningDate, 112) AS Style112
FROM Account AS A;
```

Result Set

	Style112
1	20070329
2	20090613
3	20060413
4	20151025
5	20100619
6	19850331

The first parameter specifies the resulting data type, the second parameter is any expression including, but not limited to, a column name, a string literal, another function, or two concatenated fields.

The optional style parameter can be used with certain data types and specifies the style of the output. Style are available for the following data types:

- Date and time
- Float and real
- Money and smallmoney
- Xml
- Binary

In the CONVERT example above, a style of 112 returns the four-digit year, followed by the two-digit month, and then the two-digit day. Although there are exceptions, style numbers less than 100 provide a two-digit year, while those greater than or equal to 100 display a four-digit year (including the century).

With date styles, it is important to remember that SQL uses the year 2049 as the century cut-off point. If a year is entered as the two-digit year of 49, it will be interpreted as 2049, but 50 will be interpreted as 1950. It is best practice to include four-digit years to avoid ambiguity.

	You can find more information on the conversion styles, exceptions, rules, along with numerous conversion samples at https://docs.microsoft.com/en-us/sql/t-sql/functions/cast-and-convert-transact-sql .

Choosing the correct data type and size is very important. If your destination data type is not large enough, SQL Server may truncate the data without any errors or warnings.

The PARSE Function

The parse function's added flexibility over CAST or CONVERT is the ability to assign a culture parameter which defines the order in which the data in the string should be interpreted along with what symbols should be used. As mentioned earlier, due to additional overhead associated with PARSE, only use this function when converting string values to date/time or number data, particularly when a cultural reference is needed to properly interpret the data.

For example, when a date is formatted according to the rules of a culture setting that is different from the one defined on the SQL Server operating system, the conversion fails. But if the proper culture is defined with PARSE, the conversion succeeds. The Chapter 08 Inline Sample Scripts.sql file includes examples of scripts that both fail before adding the culture definition and then succeed after. The successful script is listed here:

Syntax

```
PARSE ( string_value AS data_type [ USING culture ] )
```

Sample

```
SELECT PARSE('2 978,15 €' AS money USING 'fr-FR');
```

The STR Function

The STR function allows you to define the total length of the string being returned (including signs, spaces, decimal point, and digits) and the number of digits to the right of the decimal place.

Sytnax

```
STR ( float_expression [ , length [ , decimal ] ] )
```

Sample

```
DECLARE @numeratorvar float = 2
    , @denominatorvar float = 3
    , @floatvar float;

SET @floatvar = @numeratorvar / @denominatorvar;
SELECT @floatvar, STR(@floatvar, 20, 3);
```

 Copyright © 2018 Ann L. Weber

Result Set

	(No column name)	(No column name)
1	0.666666666666667	0.667

Try It 9 – Explicit Conversions

In this exercise, you will write and execute a series of short queries to understand how explicit conversions work. If needed, the queries can be opened from **\Chapter 03 Functions\Try It Exercises\Try It 9 - Explicit conversion.sql**.

1. Open a new query window and save the script as **ExplicitConversion.sql** to the **\Student Files** folder.
2. Type and execute the following query to convert the OpeningDate column from the Account table to a variable character field of 20 characters. Use style 112 and alias the column as [Style 112 ISO].

```
SELECT AccountID
    , CONVERT(varchar(20), OpeningDate, 112)
         AS [Style 112 ISO]
FROM Account;
```

3. Copy the column with the CONVERT function and change the style to 12 in both the function and the alias name. The query should now look like the command below:

```
SELECT AccountID
     , CONVERT(varchar(20), OpeningDate, 112)
                     AS [Style 112 ISO]
     , CONVERT(varchar(20), OpeningDate, 12)
                     AS [Style 12 ISO]
FROM Account;
```

4. Copy the converted columns to add two more columns. One for style 107 using a data type of varchar(20). Name this column [Mon dd yyyy]. The final column will be for style 12 with a datatype of varchar(4), naming the column [Truncated ISO] as follows:

```
SELECT AccountID
     , CONVERT(varchar(20), OpeningDate, 112)
          AS [Style 112 ISO]
     , CONVERT(varchar(20), OpeningDate, 12)
          AS [Style 12 ISO]
     , CONVERT(varchar(20), OpeningDate, 107)
          AS [Mon dd yyyy]
     , CONVERT(varchar(4), OpeningDate, 12)
          AS [Truncated ISO]
```

Copyright © 2018 Ann L. Weber

```
FROM Account;
```

5. Notice the results and the truncated data in the final column. Be careful with styles and data types. Save your queries and close the query window. Leave SSMS open.

TRY_PARSE, TRY_CAST, and TRY_CONVERT

Starting in SQL Server 2012, three new functions were added to avoid error messages that occur when the data to be converted fails the conversion. When you add TRY_ to the beginning of any of these function names, the function returns NULL instead of producing an error when the conversion fails.

Try It 10 – TRY_ functions

By performing the steps in the following Try It exercise, you will practice using the TRY_ functions and see how they differ from their counterparts.

1. Open the \Chapter 03 Functions\Try It Exercisese\Try It 10- TRY_ Functions Starter.sql file.
2. Click **File** | **Save** (or click the Save icon), and then browse to the **\Student Files** folder. Click **Save** to save the file in the new location.
3. Review the queries under Step #3, and then execute them. These queries will create a new table that we will use for this Try It. Review the results from the SELECT statement.
4. Try and run the command under Step #4. What happened? Why?
5. Try and run the command under Step #5. What happened? Why?
6. Under Step #6 write a single query with TRY_CONVERT that will return NULL for the Birthdate and JoinAge values that cannot be converted, while returning the properly formatted values for the other rows. The final query is located in the Try It 10 – TRY_ Functions Answer.sql file in the \Chapter 03 Functions\Try It Exercises folder. The result set is shown below. Add appropriate aliases to the column expressions.

	TestRowID	Birthdate	JoinAge
1	1	2000-01-15	15
2	2	1945-05-05	45
3	3	NULL	NULL
4	4	NULL	NULL
5	5	NULL	20

Figure 54: Results Set

7. If time permits, under Step #7, use a WHERE clause and the TRY_PARSE function to use the fr-FR culture settings to return the Birthdate column as a date

 Copyright © 2018 Ann L. Weber

datatype for rows with TestRowIDs of 4 or 5. The query can be found below and in the Try It 10 – TRY_ Functions Answer.sql answer file.

```
SELECT TestRowID
   , TRY_PARSE(TC.Birthdate AS date USING 'fr-FR')
       AS Birthdate
FROM TestConversions AS TC
WHERE TestRowID IN (4,5)
;
```

8. Execute the line with the DROP TABLE command under Step #8 to clean up the table we created for this Try It.
9. Save your completed script and close the current query window.

Chapter 4 - Handling NULL Data

In this chapter:

NULL vs blank
= vs IS NULL
ISNULL function
COALESCE
Concatenating NULL data
Chapter 4 Lab
Answers to Exercises

Files needed:

- \Chapter 04 Nulls\Inline Samples
- \Chapter 04 Nulls \Try It Exercises
- \Chapter 04 Nulls \Labs\

	Answer files can be found in the \Chapter 04 Nulls\Try It Exercises folder.

 Copyright © 2018 Ann L. Weber

NULL vs blank

Working with NULL fields can be a bit tricky. A NULL entry is not a space; rather, it represents a concept of unknown rather than empty. You cannot compare unknown values and many functions ignore or behave differently when a NULL value is involved. Because of this behavior, there are special functions and key words to use when working with NULL data.

For example, if you concatenate first, middle, and last names and some of the middle names are NULL, the result of the expression will be NULL. If you perform a count on a column with NULL values in some of the rows, the count will only include the number of rows with non-NULL values. This count will include fields with a space or an empty, but not those stored specifically as NULL.

Try It 1 – Working with NULL Data

In this exercise you will explore some of the behavior that you will see when working with NULL data.

1. In SSMS, use the Open File folder icon to open **\Chapter 04 Nulls\Try It Exercises\Try It 1 - Working with NULL Data.sql** .
2. Execute the commands under the **Step #2** comment to change the current database context and create a new table to test how NULL behaves.
3. Execute the INSERT command under the **Step #3** comment to add four rows to the newly created table.
   Note: If there are red squiggly lines underneath the table name in the INSERT statement, you can type **Ctrl + Shift + R** to refresh the local IntelliSense cache. Alternative, you can click **Edit | IntelliSense | Refresh Local Cache** on the menu.
4. Execute the SELECT statement under **Step #4**. Pay particular attention to the values in column 2, noticing that only one row has a value of NULL.
5. Execute the SELECT statement under **Step #5**. Notice that the new column in the first row is NULL, but the rest of the rows show the concatenated results. This is because NULL concatenated with anything else is NULL. Additionally, notice the difference in the results in rows 2 and 3. Row 2 has an empty string in col2, while row 3 has a space. That is why the 'c' is farther to the right than the 'b'.
6. Execute the SELECT statement under **Step #6**. Note that the row with the NULL is ignored while the empty string is counted.
7. Run the DROP TABLE command under **Step #7** to clean up the newly created table.
8. Close the query window without saving. Leave SSMS open for the next Try It exercise.

You will work more with aggregate functions in **Chapter 5 Aggregating and Grouping Data**.

	When you are looking at your data and not seeing the expected results, it is important to look for and test to see if there are NULL fields, empty character strings, or extra spaces that are changing the comparison or expression results.

= vs IS NULL

The first thing you need to understand when working with NULL values is how to return rows based on whether or not a field is populated. Although versions of SQL Server up to and including SQL 2017 allow you to default to ANSI settings, this feature has been deprecated for future versions.

When using the default ANSI NULL settings (and most common settings since 2000 or earlier), SQL does not allow you to locate NULL values using an equal sign as the comparison operator. Rather than using the equals sign "=", use IS NULL. To locate values that are not NULL, use IS NOT NULL.

Syntax

```
WHERE column_name IS NULL
```

Sample

```
SELECT *
FROM Customer
WHERE Birthdate IS NULL
```

	CustomerID	FirstName	MiddleName	LastName	Birthdate	StreetAddress	City	StateProvinceCode	CountryCode	ZipCode
1	71	Jimmy	K	Dimauro	NULL	1157 Par Drive	San Luis Obispo	*M	US	93401
2	83	Alfred	B	Sane	NULL	771 Kelly Street	Gastonia	NC	US	28052
3	84	Dinah	J	Barber	NULL	4815 Cody Ridge Road	Duncan	OK	US	73533
4	87	Isabel	H	Peck	NULL	2829 Adamsville Road	Laredo	TX	US	78040
5	95	Bruce	M	Bassett	NULL	1814 University Street	Seattle	WA	US	98101
6	106	Linda	E	Mattocks	NULL	4014 Elliot Avenue	Seattle	WA	US	98101
7	139	Wade	B	Pulido	NULL	4174 Farm Meadow Drive	Dewey	AZ	US	86327
8	162	Jimmy	NULL	Coleman	NULL	4896 Broadway Street	Hilton Head	SC	US	29928

Figure 55: Partial Results Set

	If you use an equal sign in place of IS NULL, you will not get an error, but the query will not return any data. This is not a syntax error; it is because you can't equate something to nothing.

Try It 2 – Searching for NULLs

In this practice you will review searching for NULL values within a field. The Customer table includes NULL values in the MiddleName field. You will locate both records where the middle name is NULL and

 Copyright © 2018 Ann L. Weber

also records where the middle name is populated with a value. The partial result set for Step 4 is shown below.

	CustomerID	FirstName	MiddleName	LastName
1	2	Shaina	NULL	Adams
2	3	Bonnie	NULL	Spearman
3	7	Jerry	NULL	Warner
4	10	William	NULL	Ku
5	12	Shantel	NULL	Phipps
6	14	Dorothy	NULL	Morrison
7	15	Ellen	NULL	Buck
8	16	Lisa	NULL	Ward

Figure 56: Partial Results Set

1. Open a new query window and set the current database context to RetailBankingSample.
2. Click **File | Save** (or click the Save icon). Browse to the **\Student Files** folder, change the File name to **SearchingForNulls.sql**, and then click **Save**.
3. Try using the following command to use an equal sign to return rows with NULL values in the MiddleName columns of the Customer table. Your query should retrieve the CustomerID, FirstName, MiddleName, and LastName from the Customer table. **0 rows** will be returned.

```
SELECT C.CustomerID, C.FirstName, C.MiddleName
     , C.LastName
FROM Customer AS C
WHERE C.MiddleName = NULL;
```

4. Modify your query to return all rows where the middle name is NULL as shown below. **90 rows** should be returned.

```
SELECT C.CustomerID, C.FirstName, C.MiddleName
     , C.LastName
FROM Customer AS C
WHERE C.MiddleName IS NULL;
```

5. Our next goal is to return all rows with data in the MiddleName field. Change the where clause to match the following code. **210 rows** should be returned.

```
SELECT C.CustomerID, C.FirstName, C.MiddleName
     , C.LastName
FROM Customer AS C
WHERE C.MiddleName IS NOT NULL;
```

6. Save and close your query window.

ISNULL function

When working with NULL values, two important functions are ISNULL and COALESCE. The ISNULL function replaces all NULL values in a column with a specified value. For example, when you need to compute an Average value for a column, any NULL values in the column are ignored. If you have 10 rows in a table, but only 8 of those rows contain data in a specified column, calculating the average value of that column will divide the total value by 8, not 10. If NULL values are stored, but you want to treat them as 0, use the ISNULL function.

Syntax

```
ISNULL(Column_or_expression, replacement _ value)
```

Sample

```
SELECT CustomerID, ISNULL(MiddleName, '') AS MiddleName
FROM Customer;
```

The sample above will replace any NULL values in the MiddleName columns with an empty string. Include a space inside the single quotes to add a space, such as when concatenating. If you do not use ISNULL in this example and try to concatenate CustomerID and MiddleName, any rows with NULL in the MiddleName column will return NULL.

COALESCE

Like ISNULL, COALESCE allows you to replace a NULL value with a data value. The difference is that COALESCE lets you define a list of expressions to return instead of NULL values, including fixed values. For example, in the PhishingSample database, the third column in the result set shown in Figure 57, InitialActionTime, will return the first non-NULL value of either TimeReportedSpam, TimeOpened, or a fixed value of '99991231' ('12/31/9999') for each row.

Syntax

```
COALESCE ( expression [, …n] )
```

Samples

Sample 1 similar to ISNULL

```
SELECT CustomerID, COALESCE(MiddleName, '') AS MiddleName
FROM Customer;
```

Sample 2 additional functionality – PhishingSample Database

```
USE PhishingSample;
SELECT L.LookupID, L.UserID
   , COALESCE(L.TimeReportedSpam
                         , L.TimeOpened, '99991231')
             AS InitialActionTime
```

 Copyright © 2018 Ann L. Weber

```
FROM [Lookup] AS L
;
```

	LookupID	UserID	InitialActionTime
1	1	1001436	2018-02-15 17:26:29.000
2	2	1001223	2018-02-15 14:48:55.000
3	3	1001381	2018-02-15 15:52:18.000
4	4	1000512	9999-12-31 00:00:00.000
5	5	1000585	2018-02-15 14:04:45.000
6	6	1000936	2018-02-15 14:03:15.000
7	7	1001122	2018-02-15 14:52:16.000
8	8	1001818	2018-02-15 13:15:23.000

Figure 57: Partial Results Set

Unless your specific situation dictates otherwise, list the most fully populated column first for optimal performance of the COALESCE expression. Once the server finds a non-NULL value, it places that value in the result set without processing the rest of the function options.

Try It 3 – COALESCE

In this exercise, you will practice using the COALESCE expression to return employee pay information from the Employee table.

In the RetailBankingSample database employees' pay is stored in one of three ways; annual salary, hourly pay, or a fixed rate paid for a specific number of pay events. An example of an employee who gets paid for a specific number of pay events would be an auditor who gets paid per audit. To get an annual rate for hourly employees, multiply their pay rate by 2080. An employee will never have data in more than one pay type.

Figure 58 below shows a partial result set from the final query in this Try It.

	EmployeeID	FirstName	LastName	AnnualPay
1	1	William	Diamond	999999.00
2	2	Aurelio	Bryant	164385.00
3	3	Jill	Moore	60267.00
4	4	Howard	Yeager	39520.00
5	5	Nancy	Peery	45760.00
6	6	Cathryn	Edwards	45760.00
7	7	Sylvia	Flowers	29120.00
8	8	Michell	Brown	37440.00

Figure 58: Partial Results Set

1. Open a new query window.
2. Click **File | Save** (or click the Save icon). Browse to the **\Student Files** folder, change the File name to **COALESCE.sql**, and then click **Save**.
3. Write a query that returns an employee's EmployeeID, FirstName, and LastName. Execute your query. The Try It 3 – COALESCE.sql file includes the starter query if you need help.
4. Use a COALESCE expression to add a column aliased as AnnualPay. Use the information from the Try It introduction to help you determine the required expressions. The following list includes information for determining the annual pay. If needed, refer to the query below.
   a. The Salary field already represents an annual number.
   b. The HourRate * 2080 will provide an annual number for hourly employees.
   c. The FixedRate * FixedRateAnnualCount provides the annual pay for fixed rate employees.

```
SELECT E.EmployeeID, E.FirstName, E.LastName
  ,COALESCE(E.Salary, E.HourlyRate * 2080
          , E.FixedRate * E.FixedRateAnnualCount)
            AS AnnualPay
FROM Employee AS E
;
```

5. Execute the query.
6. Save your query and close the query window. Leave SSMS open for the next Try It.

Concatenating NULL data

As you saw in the first Try It exercise, if any one field is NULL when you concatenate multiple fields, the result is NULL, or Unknown. An alternative to the default functionality is to use the CONCAT function. The CONCAT function concatenates any number of strings and replaces NULL values with an empty string instead of returning Unknown.

 Copyright © 2018 Ann L. Weber

Depending on your preferences and on your organization's standards, you can switch back and forth between using the plus sign (+) and the CONCAT function for concatenation.

Syntax

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

Sample

```
SELECT CONCAT(RTRIM(E.LastName), ', ', E.FirstName) AS
FullName
FROM Employee AS E
;
```

	FullName
1	Diamond, William
2	Bryant, Aurelio
3	Moore, Jill
4	Yeager, Howard
5	Peery, Nancy
6	Edwards, Cathryn
7	Flowers, Sylvia
8	Brown, Michell

Figure 59: Partial Results Set

Prior to introducing the CONCAT function, query writers had to use a combination of CASE, ISNULL, or other functions to concatenate multiple fields that included NULL values. Queries were even more complex when adding commas or spaces between fields. In the following Try It you will use the CONCAT function to concatenate a full name field without returning extra spaces when the middle name is NULL.

Try It 4 – Concatenating with NULLs

In this exercise you will use the CONCAT function and the plus sign (+) to concatenate first, middle, and last names from the Customer table together. You will work through several partial solutions first to help you better understand how the CONCAT function works. The final result should look similar to Figure 60.

	FullName
1	Thomas F Dudley
2	Shaina Adams
3	Bonnie Spearman
4	David M New
5	Marilyn C Whitworth
6	Shaun A Jones
7	Jerry Warner
8	Elizabeth N Mitchell

Figure 60: Partial Results Set

1. Open a new query window.
2. Click **File | Save** (or click the Save icon). Browse to the **\Student Files** folder, change the File name to **COALESCE.sql**, and then click **Save**.
3. Type the following query to concatenate the FirstName, MiddleName, and LastName fields from the Customer table into a column aliased as FullName.

```
SELECT CONCAT(C.FirstName, C.MiddleName, C.LastName)
          AS FullName
FROM Customer AS C
;
```

   Note: Because the FirstName field is a fixed character field while the MiddleName is a variable length field, there are a lot of spaces after shorter first names and the middle initial is immediately before the LastName.

4. Add an RTRIM function to the first and last name fields to remove the extra spaces caused by the fixed character data type as shown below.

```
SELECT CONCAT(RTRIM(C.FirstName),
              C.MiddleName, RTRIM(C.LastName))
          AS FullName
FROM Customer AS C
;
```

5. Notice that the result is now all one word. Add spaces into the CONCAT function so that there is one space between first, middle, and last names as shown below.

```
SELECT CONCAT(RTRIM(C.FirstName), ' '
                               , C.MiddleName, ' '
                               , RTRIM(C.LastName)
                             ) AS FullName
FROM Customer AS C
;
```

 Copyright © 2018 Ann L. Weber

6. Depending on your font, you may or may not notice an extra space between the first and last names for people without a middle name. Instead, you could use the plus (+) sign to concatenate the middle name and the trailing space. Your result set will not have the extra space because the NULL middle name + ' ' will result in NULL, and the CONCAT function turns a NULL into an empty string.

```
SELECT CONCAT(RTRIM(C.FirstName), ' '
               , C.MiddleName + ' '
               , RTRIM(C.LastName)
              ) AS FullName
FROM Customer AS C
;
```

7. Save and close any open queries. Leave SSMS open for the lab.

Copyright © 2018 Ann L. Weber

Chapter 5 - Aggregating and Grouping Data

In this chapter:

Aggregate functions
GROUP BY
HAVING
HAVING vs WHERE
Overview ROLLUP and CUBE
OVER with Aggregates
OVER with Ranking Functions
Chapter 5 Lab
Answers to Exercises

Files needed:

- \Chapter 05 Aggregates\Inline Samples
- \Chapter 05 Aggregates \Try It Exercises
- \Chapter 05 Aggregates \Labs\
- \Student Files

Some of the Try It exercises in this chapter build on one another. They independent of other chapters. Completed Try It queries can be found in the \Chapter 05 Aggregates\Try It Exercises folder.

 Copyright © 2018 Ann L. Weber

SQL Server provides grouping and aggregating functionality to allow users to summarize and analyze data, uncover trends and commonalities within the underlying data, and to better understand the data that has been collected.

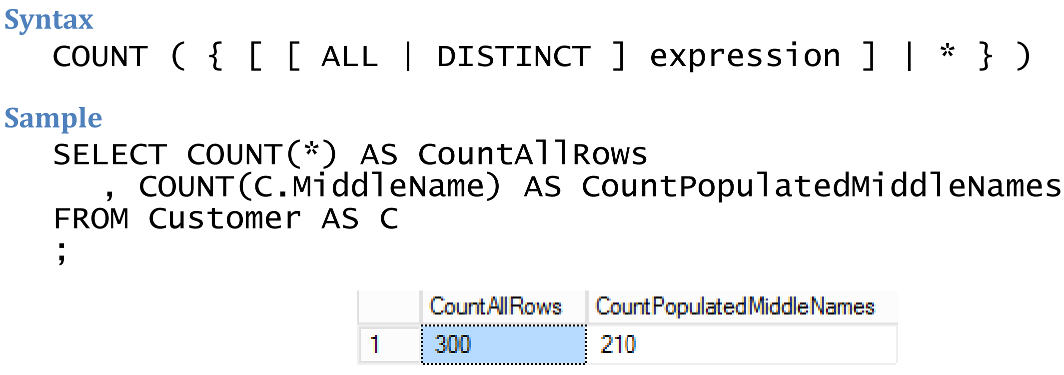
Aggregate functions

Like the scaler functions you learned about in Chapter 3, aggregate functions accept a parameter, typically a column or expression representing a set of values, and return a single value.

COUNT and COUNT_BIG

COUNT and COUNT_BIG are a little different from the other aggregate functions because they have an optional input of an asterisk (*) instead of the column/expression option. The asterisk tells the server to count the number of rows (records) in the result set rather than counting values in a specific column. If the database does not have any NULL values in the specified column and the DISTINCT keyword is not used, then COUNT(*) and COUNT(expression) will return the same count. On the other hand, when NULL values are present or the DISTINCT keyword is used, COUNT (*) and COUNT (expression) will return different counts.

The only difference between COUNT and COUNT_BIG is the datatype that is being returned. COUNT returns the int data type, while COUNT_BIG returns the value with the bigint data type. If your table uses the BIGINT data type for the key column because you are expecting to insert more than 2 billion rows in the table, then you should use COUNT_BIG function to support all of the rows.



Syntax

```
COUNT ( { [ [ ALL | DISTINCT ] expression ] | * } )
```

Sample

```
SELECT COUNT(*) AS CountAllRows
    , COUNT(C.MiddleName) AS CountPopulatedMiddleNames
FROM Customer AS C
;
```

	CountAllRows	CountPopulatedMiddleNames
1	300	210

Figure 61: Results

Because 90 rows contain a NULL value in the MiddleName field of the Customer table, the query above returns 210 in the CountPopulatedMiddleNames derived column vs the count of 300 returned from the COUNT(*) portion of the query. You can retrieve the number of populated middle names by using COUNT(*) and a WHERE clause that only returns rows where the MiddleName IS NOT NULL. To count the records where the middle name is NULL, use IS NULL in the WHERE clause.

MIN, MAX, SUM, AVG

The functions that return the minimum, maximum, mathematical sum, and mathematical average all use the same general syntax.

Syntax

```
MIN ( [ ALL | DISTINCT ] expression )
```

Sample

```
SELECT MIN(AT.Amount) AS Minimum
   , MAX(AT.Amount) AS Maximum
   ,AVG(AT.Amount) AS Average
   ,SUM(AT.Amount) AS Total
FROM AccountTransaction AS AT
;
```

	Minimum	Maximum	Average	Total
1	-10375.56	11302.14	-326.0594	-26467550.89

Figure 62: Results

While SUM and AVG only work with numeric data types, MIN and MAX allow expressions of many data types including numeric, char, varchar, nchar, nvarchar, uniqueidentifier, or datetime columns.

When working with dates, MIN will return the date farthest into the past and MAX will return the latest date available in the dataset, even if it is in the future.

CHECKSUM_AGG, GROUPING, GROUPING_ID, STDEV, STDEVP, VAR, and VARP are beyond the scope of this class. You can find additional information about these functions at https://docs.microsoft.com/en-us/sql/t-sql/functions/aggregate-functions-transact-sql?view=sql-server-2017.

Aggregating and Nulls

Aggregate functions that have a column or column-based expression passed to them ignore all NULL values. As discussed earlier, because COUNT(*) is counting rows (records) in the defined set rather than values in a specific field, COUNT(*) is not affected by NULL values. However, COUNT(MiddleName) will only count the number of values that are NOT NULL.

Depending on the aggregation you are using, ignoring NULL values may have unwanted consequences, such as when determining an average or performing a count. Skipping NULL values in a SUM operation doesn’t affect the result in any way.

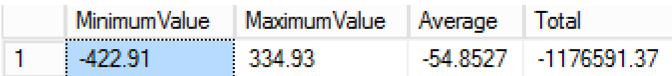
The note that tells you NULL values have been ignored is displayed on the Messages tab, making it easy to miss.

Try It 1 – Aggregate Functions

In this exercise, you will write a query that returns the minimum, maximum, average, and sum for all amounts listed in the LoanTransaction table. The results for this query can be seen in Figure 63. You will

 Copyright © 2018 Ann L. Weber

add a second query that returns the count of all employees, the count of the number of employees who are paid with an annual salary, the count of employees paid an hourly rate, and the count of employees paid a fixed rate.



	MinimumValue	MaximumValue	Average	Total
1	-422.91	334.93	-54.8527	-1176591.37

Figure 63: Results

1. If necessary, open SQL Server Management Studio (SSMS).
2. Click the New Query button in the General toolbar to open a new Query Editor tab.
3. Click **File | Save** (or click the Save icon). Browse to the **\Student Files** folder, change the File name to **AggregateFunctions.sql**, and then click **Save**.
4. Either select RetailBankingSample from the database drop-down list in the SQL Editor toolbar, or type and execute the following SQL command. Press F5 or click the Execute button to execute the script.

```
USE RetailBankingSample;
```

5. Type and execute the following query to return the minimum, maximum, average, and sum for all of the amounts listed in the CreditTransaction table

```
SELECT MIN(LT.Amount) AS MinimumValue
     , MAX(LT.Amount) AS MaximumValue
     , AVG(LT.Amount) AS Average
     , SUM(LT.Amount) AS Total
FROM LoanTransaction AS LT
;
```

6. In the RetailBankingSample database, employees are paid in one of three ways. If their pay is based on an annual salary, the amount is in the Salary column. If the employee is paid an hourly rate, their rate is in the HourlyRate column. Some employees are only paid when they perform a certain task, with a fixed rate per task accomplished. The "per task" rate is in the FixedRate column. When the column does not apply to the employee, NULL values are used. Type and execute the following query to find the total number of employees, and the numbers that are in the three different pay catagories. The result set is shown in Figure 64.

	EmployeeCount	SalaryEmployeeCount	HourlyEmployeeCount	FixedRateEmployeeCount
1	50	32	16	2

Figure 64: Results

```
SELECT  COUNT(*) AS EmployeeCount
     , COUNT(E.Salary) AS SalaryEmployeeCount
```

```
        , COUNT(E.HourlyRate) AS HourlyEmployeeCount
        , COUNT(E.FixedRate) AS FixedRateEmployeeCount
   FROM Employee AS E
   ;
```

7. Save your query script and leave the tab and SSMS open for the next exercise.

GROUP BY

When working with aggregates, if you want to also include columns or expressions in the SELECT or ORDER BY clauses that are not aggregated, the Group By clause is required. If you forget to add any non-aggregated columns in the SELECT clause to a GROUP BY clause, you will receive a message similar to the following:

> "Column 'LoanTransaction.AcctID' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause."

A similar error message is returned when the WHERE clause column is not in a GROUP BY clause.

Syntax

```
GROUP BY {
      column-expression
    | ROLLUP ( <group_by_expression> [ ,...n ] )
    | CUBE ( <group_by_expression> [ ,...n ] )
    | GROUPING SETS ( <grouping_set> [ ,...n ]  )
    | () --calculates the grand total
} [ ,...n ]
```

Sample

```
  SELECT AT.AcctID
     , MIN(AT.Amount) AS Minimum
     , MAX(AT.Amount) AS Maximum
     ,AVG(AT.Amount) AS Average
     ,SUM(AT.Amount) AS Total
  FROM AccountTransaction AS AT
  GROUP BY AT.AcctID
  ;
```

	AcctID	Minimum	Maximum	Average	Total
1	23	-8571.61	9151.06	-239.829	-232154.51
2	69	-6923.09	8264.60	-349.0665	-379784.42
3	92	-8770.54	11086.28	-285.2833	-332069.78
4	138	-5254.13	4567.16	68.6475	4736.68
5	161	-8428.85	6912.88	-285.8009	-85168.69
6	278	-6973.78	7481.17	-217.0762	-120911.45
7	95	-9480.57	7139.55	-479.2108	-206060.68
8	72	-8141.95	8774.60	-235.4501	-178000.29

Figure 65: Partial Results Set

 Copyright © 2018 Ann L. Weber

Try It 2 – GROUP BY

In this exercise you will modify the queries from the previous practice to include additional information in the SELECT clause and add the necessary GROUP BY statements to support the additional columns.

1. If the **AggregateFunctions.sql** file is not open from the previous Try It exercise, click **File | Open | File** (or click the Open File icon) and browse to the **\Student Files\ AggregateFunctions.sql** file.

   Note: If you did not complete the previous Try It exercise, browse to **\Chapter 05 Aggregates\Try It Exercises\ Try It 2 – GROUP BY Starter.sql**.
2. Verify that the RetailBankingSample database is active.
3. Modify the query from step 5 of Try It 1 that returned the MIN, MAX, AVG, and SUM of the Amount column from the Loan Transaction table to retrieve these values on a per account basis. A partial result set is displayed below. The completed query is shown below:

	AcctID	MinimumValue	MaximumValue	Average	Total
1	46	-87.55	48.36	-24.3375	-4307.75
2	215	-245.85	213.52	-21.6758	-1148.82
3	192	-91.87	62.34	-16.6873	-1084.68
4	209	-238.18	206.85	-21.3254	-3646.66
5	169	-202.83	137.64	-37.1464	-4346.14
6	181	-166.61	67.69	-57.5525	-11107.65
7	195	-281.08	114.19	-103.0438	-28337.05
8	244	-174.18	151.27	-15.5369	-761.31

Figure 66: Partial Results Set

```
SELECT LT.AcctID
   , MIN(LT.Amount) AS MinimumValue
   , MAX(LT.Amount) AS MaximumValue
   , AVG(LT.Amount) AS Average
   , SUM(LT.Amount) AS Total
FROM LoanTransaction AS LT
GROUP BY LT.AcctID
;
```

4. Modify and execute the query from Step 6 in Try It 1 so that it returns the defined counts per Title in the Employee Table. The result set is shown below. You query should look similar to the one below.

	Title	EmployeeCount	SalaryEmployeeCount	HourlyEmployeeCount	FixedRateEmployeeCount
1	Auditor	2	0	0	2
2	Banker	25	15	10	0
3	Manager	2	2	0	0
4	Supervisor	8	8	0	0
5	Teller	13	7	6	0

Figure 67: Results

```
SELECT  Title
     , COUNT(*) AS EmployeeCount
     , COUNT(E.Salary) AS SalaryEmployeeCount
     , COUNT(E.HourlyRate) AS HourlyEmployeeCount
     , COUNT(E.FixedRate) AS FixedRateEmployeeCount
FROM Employee AS E
GROUP BY Title
;
```

5. Save your query, and leave both SSMS and the query tab open for the next Try It.

HAVING

The having clause limits the result set based on the results of an aggregated value. For example, you can use a HAVING clause to find all AccountIDs that have a positive sum for the amount in all rows for that AccountID.

Syntax

```
[ HAVING <search condition> ]
```

Sample

```
SELECT AT.AcctID
   , MIN(AT.Amount) AS Minimum
   , MAX(AT.Amount) AS Maximum
   ,AVG(AT.Amount) AS Average
   ,SUM(AT.Amount) AS Total
FROM AccountTransaction AS AT
GROUP BY AT.AcctID
HAVING SUM(AT.Amount) > 0
;
```

	AcctID	Minimum	Maximum	Average	Total
1	138	-5254.13	4567.16	68.6475	4736.68
2	94	-8749.95	6200.06	116.8622	7128.60

Figure 68: Results

 Copyright © 2018 Ann L. Weber

Try It 3 – HAVING Clause

In this exercise, you will add a restriction to the query that counts employees of different pay types by Title from the previous Try It. You will restrict this query to return only employee types that have a total employee count greater than 10 in that category.

1. If the **AggregateFunctions.sql** file is not open from the previous Try It exercise, click **File | Open | File** (or click the Open File icon) and browse to the **\Student Files\ AggregateFunctions.sql** file.

   Note: If you did not complete the previous Try It exercise, browse to **\Chapter 05 Aggregates\Try It Exercises\ Try It X - Having Starter.sql**.
2. Verify that the RetailBankingSample database is active.
3. Add code to restrict the output to ONLY include employee titles with more than 10 total employees. This code will be added to the final query that counts employees of different pay types by title, as completed in Try It 2, Step 4. **2 rows** should be returned as shown in Figure 69. The full query is below the result set.

	Title	EmployeeCount	SalaryEmployeeCount	HourlyEmployeeCount	FixedRateEmployeeCount
1	Banker	25	15	10	0
2	Teller	13	7	6	0

Figure 69: Results

```
SELECT  Title
      , COUNT(*) AS EmployeeCount
      , COUNT(E.Salary) AS SalaryEmployeeCount
      , COUNT(E.HourlyRate) AS HourlyEmployeeCount
      , COUNT(E.FixedRate) AS FixedRateEmployeeCount
FROM Employee AS E
GROUP BY Title
HAVING COUNT(*) > 10
;
```

4. Save your query and close the current query tab, but leave SSMS open for the following Try It exercise.

HAVING vs WHERE

It is sometimes difficult to remember when to use HAVING and when to use WHERE. Remember: the WHERE clause limits the result set based on data that exists in the table, and the HAVING clause limits the result set based on an aggregate and the GROUP BY clause.

The following sample uses the WHERE clause to limit the rows to those that have a TransactionType of “Direct Deposit” in the table and the HAVING clause to only return Accounts with a SUM of the values in the Amount column greater than 400,000.

Sample

```
SELECT AT.AcctID
   , MIN(AT.Amount) AS Minimum
   , MAX(AT.Amount) AS Maximum
   ,AVG(AT.Amount) AS Average
   ,SUM(AT.Amount) AS Total
FROM AccountTransaction AS AT
WHERE TransactionType = 'Direct Deposit'
GROUP BY AT.AcctID
HAVING SUM(AT.Amount) > 400000
;
```

	AcctID	Minimum	Maximum	Average	Total
1	92	0.47	8495.34	1995.1043	405006.18
2	235	8.21	6670.07	1927.6535	1015873.41
3	272	16.39	7590.38	2019.7692	405973.61
4	135	3.23	7376.62	1984.8917	1014279.70
5	259	0.29	7817.79	2183.3537	932292.04
6	21	13.14	7539.44	2059.3176	677515.50
7	58	5.57	8292.53	2139.6372	828039.62
8	7	7.64	8050.04	2021.8272	1002826.31

Figure 70: Partial Results Set

The following row counts demonstrate the effect of the different clauses on the size of the result set:

- 93 rows returned - No WHERE or HAVING clauses
- 93 rows returned - WHERE clause only
  Note: Aggregate values only include direct deposit values and are very different from when the WHERE clause is not included
- 0 rows returned - HAVING clause only
- 17 rows returned - Both WHERE and HAVING clauses

Try It 4 – HAVING vs WHERE

In this exercise, you will write a query that incorporates the GROUP BY techniques that you have learned in this chapter to produce the result set shown in Figure 71. The requirements for this query are as follows:

- Retrieve the AccountID from the LoanTransaction table.
- Retrieve the total of all values in the Amount column per AccountID aliased as TotalInterest.
- Limit the result set to include only rows with a TransactionType of **Interest.**
- Limit the result set to include only rows with a TotalInterest value over 10,000.

 Copyright © 2018 Ann L. Weber

	AcctID	TotalInterest
1	209	16836.82
2	195	10451.99
3	189	11714.75
4	118	22778.85
5	212	11813.10
6	149	12335.55
7	144	10070.00
8	67	22062.88

Figure 71: Partial Results Set

1. Add a new Query Editor tab and save your new query as **\Student Files\HAVING vs WHERE.sql**.
2. Set the database context to RetailBankingSample.
3. Work together as a group to type and execute a query that meets all the requirements defined at the beginning of the Try It Exercise. 38 rows should be returned. If you need help, the query is included in the **\Chapter 05 Aggregates \Try It Exercises\ Try It 4- HAVING and WHERE.sql** script.
4. Save and close your query, but leave SSMS open for the next Try It exercise.

Overview ROLLUP and CUBE

When you have a GROUP BY statement with an aggregate, the result set includes summary data based on the combination of the non-aggregated columns in the SELECT/GROUP BY clauses. The ROLLUP and CUBE operators provide grand and subtotal levels in addition to the more detailed summarization that occurs with GROUP BY.

When working with 3 non-aggregated columns in standard SELECT and GROUP BY clauses, the aggregated values are provided for each unique combination of the three columns, with no subtotals or totals per column. If you include AcctID, TransactionDate, and TransactionType in both the SELECT and GROUP BY clauses along with a SUM of the Amount column, and AcctID 1 has two transactions on Jan 1 and one transaction is an ATM withdrawal and the other is a Debit Card, you will get one row with the total amount for AcctID 1, January 1, TransactionType 1, and one row for AcctID 1, January 1, TransactionType 2. You will not see subtotals by AcctID or TransactionDate or TransactionType.

When the ROLLUP operator is specified, in addition to the rows received from the GROUP BY statement, the query also returns a grand total row (denoted by a NULL in each of the non-aggregated columns of the result set) and a subtotal row for each value in the first column listed in the GROUP BY clause. This row is represented by a NULL in each of the other two columns listed in the GROUP BY clause. Note that the NULL values returned from ROLLUP do not mean “the absence of data”, as is usually the case with NULL, but rather denotes a total, either a grand total or subtotal.

When the CUBE operator is specified, the result set includes the rows returned by a standard GROUP BY, the grand total also returned by ROLLUP, and the subtotals for all columns in the GROUP BY list, not just the first column.

GROUPING SETS offer greater flexibility than ROLLUP and CUBE for creating subtotals by different groups, but are beyond the scope of this course. You can read more about GROUPING SETS at https://docs.microsoft.com/en-us/sql/t-sql/queries/select-group-by-transact-sql?view=sql-server-2017.

Syntax

```
GROUP BY {
      column-expression
    | ROLLUP ( <group_by_expression> [ ,...n ] )
    | CUBE ( <group_by_expression> [ ,...n ] )
```

The following three samples are described below and include partial result sets. You can explore the results more easily be executing the queries located under the “ROLLUP and CUBE” comments in the **\Chapter 05 Aggregates\Inline Samples\Inline Samples 05.sql** script.

The first query returns 20 rows and includes the number of employees with a particular title that report to the same person. (The ReportTo field represents the EmployeeID of the manager of the record being viewed.) There are two Managers who do not report to anyone, so the ReportTo field is NULL. This makes the results created by ROLLUP and CUBE harder to interpret but is a common situation in the real world.

Sample ROLLUP and CUBE Starter

```
SELECT E.ReportsTo, E.Title
   , COUNT(*) AS EmployeeCount
FROM Employee AS E
GROUP BY E.ReportsTo, E.Title
ORDER  BY E.ReportsTo
;
```

	ReportsTo	Title	EmployeeCount
1	NULL	Manager	2
2	1	Auditor	1
3	1	Supervisor	6
4	2	Auditor	1
5	2	Supervisor	2
6	16	Banker	1
7	16	Teller	1
8	18	Banker	4

Figure 72: Partial Results Set

 Copyright © 2018 Ann L. Weber

The second query returns 32 rows, including the 20 rows returned by the "Starter" query. This result set also includes a grand total row and a summary (subtotal) row for each EmployeeID that appears in a ReportsTo field. For example, the grand total is represented in the row with both the ReportsTo and Title columns as NULL and an EmployeeCount of 50. Additionally, you can see that EmployeeID 1 has one Auditor reporting to them and 6 Supervisors. Thus, the total number of people reporting to EmployeeID 1 is 7.

Sample ROLLUP Option

```
SELECT ReportsTo, E.Title
   , COUNT(*) AS EmployeeCount
FROM Employee AS E
GROUP BY ROLLUP (E.ReportsTo, E.Title)
ORDER BY E.ReportsTo
;
```

	ReportsTo	Title	EmployeeCount
1	NULL	Manager	2
2	NULL	NULL	2
3	NULL	NULL	50
4	1	Auditor	1
5	1	Supervisor	6
6	1	NULL	7
7	2	Auditor	1
8	2	Supervisor	2
9	2	NULL	3
10	16	Banker	1

Figure 73: Partial Results Set

In addition to the rows that ROLLUP returns, the final query with the CUBE operator returns a subtotal row for each ReportsTo and Title column, thus returning a total of 37 rows. This result set contains one additional row for each of the five Title values in the table. As stated earlier, the detail rows for the NULL ReportsTo data (Managers who don't report to anyone) returns an EmployeeCount of 2.

Sample CUBE Option

```
SELECT ReportsTo, E.Title
   , COUNT(*) AS EmployeeCount
FROM Employee AS E
GROUP BY CUBE (E.ReportsTo, E.Title)
ORDER BY E.ReportsTo, E.Title
;
```

	ReportsTo	Title	EmployeeCount
1	NULL	NULL	50
2	NULL	NULL	2
3	NULL	Auditor	2
4	NULL	Banker	25
5	NULL	Manager	2
6	NULL	Manager	2
7	NULL	Supervisor	8
8	NULL	Teller	13
9	1	NULL	7
10	1	Auditor	1
11	1	Supervisor	6
12	2	NULL	3
13	2	Auditor	1
14	2	Supervisor	2
15	16	NULL	2
16	16	Banker	1

Figure 74: Partial Results Set

The GROUPING function returns 1 when the expression passed as a parameter to the function represents a summary row that was added by ROLLUP, CUBE, or GROUPING SETS. A 0 represents the detailed aggregations created by the GROUP BY clause. You can read more about the GROUPING function at https://docs.microsoft.com/en-us/sql/t-sql/functions/grouping-transact-sql?view=sql-server-2017. Additionally, the \Chapter 05 Aggregates\Inline Samples\Extra Samples 05.sql file includes a sample of the GROUPING function added to the CUBE sample above.

OVER with Aggregates

The OVER clause was added to the SELECT statement functionality in SQL Server 2008. OVER allows you to partition and order a rowset and then apply a function (typically aggregate and ranking functions) "over" each defined "window" of data. You will see these functions referred to "window functions". These functions have nothing to do with the Windows Operating System.

When you want to look at grand total and/or subtotal aggregates in the same rows as detailed data, the OVER clause can help you get around the restrictions placed on not including columns that are not specified in a GROUP BY clause in the SELECT or ORDER BY clauses.

The OVER clause is beneficial for creating rolling (moving) totals or averages, cumulating running totals, top values per group, and more.

 Copyright © 2018 Ann L. Weber

Syntax

```
OVER (
       [ <PARTITION BY clause> ]
       [ <ORDER BY clause> ]
       [ <ROW or RANGE clause> ]
      )
```

The PARTITION BY clause divides the overall result set into smaller pieces on which the function performs. For example, if I wanted to provide an annual running total, adding each new value to the previous one, but then resetting for the beginning of each new year, partitioning by a year column or the year part of a date column would provide this functionality.

Using the same scenario, the ORDER BY clause would use the date field so that the SQL Server query engine would know where to start the aggregations. ORDER BY plays a larger role in the RANKING functions covered later in this chapter. If an ORDER BY is not stated, the order that the data is returned from the query engine is the order the values are added together. This would make no sense for a running total. Additionally, if there are tied values when ordering by the first column in the ORDER BY clause, the order will be random unless a second column is also included as in the sample below.

The ROWS or RANGE clause was added in SQL Server 2012. This clause allows you to further limit the result set to a portion of a partition and define either a fixed set of rows or a range of rows based off of the current row. For example, if your data has monthly totals and you want to return a 3-month rolling (moving) average, use the AVG aggregate and an OVER clause that defines a ROWS clause that includes the current row and the 2 months prior.

Sample

```
SELECT LT.LoanTransactionID, LT.TransactionDate, LT.Amount
   , SUM(LT.Amount)
        OVER (PARTITION BY DatePart(yy,LT.TransactionDate)
                             ORDER BY
LT.TransactionDate,LT.LoanTransactionID
                   )
   AS AnnualRunningTotal
FROM LoanTransaction AS LT
WHERE LT.TransactionType = 'Interest'
ORDER BY TransactionDate, LoanTransactionID
;
```

Note: The ORDER BY clause at the end of the SELECT statement makes the results easier to read but does NOT affect the aggregation.

	LoanTransactionID	TransactionDate	Amount	AnnualRunningTotal
1	7076	1980-12-24 00:00:00	125.50	125.50
2	7077	1981-01-24 00:00:00	125.31	125.31
3	7078	1981-02-24 00:00:00	125.12	250.43
4	7079	1981-03-24 00:00:00	124.93	375.36
5	7080	1981-04-24 00:00:00	124.74	500.10
11	7086	1981-10-24 00:00:00	123.59	1244.52
12	7087	1981-11-24 00:00:00	123.39	1367.91
13	7088	1981-12-24 00:00:00	123.20	1491.11
14	7089	1982-01-24 00:00:00	123.00	123.00
15	7090	1982-02-24 00:00:00	122.81	245.81
16	7091	1982-03-24 00:00:00	122.61	368.42
17	7092	1982-04-24 00:00:00	122.41	490.83

Figure 75: Partial Results Set

The ROWS | RANGE clause is beyond the scope of this class. You can find additional information at https://docs.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql?view=sql-server-2017 and a sample similar to the example defined above in the \Chapter 05 Aggregates\Inline Samples\Extra Samples 05.sql file.

Try It 5 – OVER with Aggregates

In this exercise, you will open and modify a query to add a column that will return the count of the number of employees from any given state. The existing query removes any rows with an unknown state which is noted as *M.

9. In SSMS, use the Open File folder icon to open **\Chapter 05 Aggregates\Try It Exercises\Try It 5 – OVER with Aggregates Starter.sql**.
10. Click **File | Save Try It 5 – OVER with Aggregates Starter.sql As**, and then browse to the **Student Files** folder. Type **Ch5TryIt5.sql** in the File name, and then click **Save**.
11. Execute the entire query and review the result set. Notice that there are a number of customers from the same states.
12. Add an aggregate to count the number of customers that exist in the same state as the current row. Use the OVER and PARTITION BY clauses. A partial result set and the completed query are provided below.

 Copyright © 2018 Ann L. Weber

	CustomerID	FirstName	LastName	City	StateProvinceCode	SameStateCustomerCount
1	6	Shaun	Jones	Birmingham	AL	6
2	69	Jacob	Williams	Mobile	AL	6
3	94	Kenneth	Nilles	Mobile	AL	6
4	107	Mary	Thelen	Birmingham	AL	6
5	140	Efren	Beaudry	Alabaster	AL	6
6	165	Michael	Noah	Montgomery	AL	6
7	215	Ricky	Garner	Wichita	AR	2
8	102	Patrick	Goebel	Pine Bluff	AR	2
9	139	Wade	Pulido	Dewey	AZ	1
10	146	Shirley	Caldwell	San Francisco	CA	20
11	147	Sylvia	Carl	Los Angeles	CA	20

Figure 76: Partial Results Set

```
SELECT C.CustomerID, C.FirstName, C.LastName
     , C.City, C.StateProvinceCode
     , COUNT(*) OVER (PARTITION BY C.StateProvinceCode)
          AS SameStateCustomerCount
FROM Customer AS C
WHERE StateProvinceCode <> '*M'
;
```

13. Save and close your query, but leave SSMS open for the next Try It exercise.

OVER with Ranking Functions

In SQL 2008 and later, the ranking functions provide an alternative to the TOP keyword and provide additional functionality for ranking and comparing rows.

There are four functions in the Ranking group of built-in functions as described below:

- ROW_NUMBER – assigns row numbers based on the ORDER BY clause in the OVER clause. If no ORDER BY clause is specified, or if there are ties within the field(s) specified in the ORDER BY clause, the next row that is returned is given the next value. With ROW_NUMBER, no numbers are repeated or skipped when ties occur.
- RANK – rows are numbered based on the ORDER BY clause in the OVER clause. When ties occur, the rank number is repeated for each row involved in the tie. The next value following the tie is assigned what the rank would be if every tied row had been assigned a unique value. For example, if three rows are tied for the third place, while the first two rows were unique, the ranks would be assigned as 1,2,3,3,3,6.
- DENSE_RANK is very similar to RANK, but returns the next available number after ties, not skipping any numbers. In the same example from above, DENSE_RANK would return 1,2,3,3,3,4.

- NTILE is different from the other Ranking functions in that it requires an integer expression to be passed as a parameter at execution time and rather than applying a row number or rank to each row, it splits the rows into “buckets” and assign a number to each bucket. Rows are assigned to the buckets in the order defined by the ORDER BY clause. If the total result set row count is not evenly divisible by the number of buckets, one extra row is added to each bucket until the remainder has been used up. For example, if I have 10,110 rows and we are breaking the data into 100 groups, the first ten groups will have 102 rows each. The rest of the groups will have 101 rows.

If there are ties in the RowNumber and NTile functions, the row assigned the next row number or NTile group is determined based on the way the server retrieved the data, not an order that you define. For example, in the result set shown below in Figure 77, when the amount is 0, the row numbers are assigned in the order that the server returns the data. Additional columns in the ORDER BY clause inside of the OVER clause, such as the OpeningDate column can help overcome the “randomness” of the number assignments for ties.

Syntax (RANK, DENSE_RANK, and ROW_NUMBER)

```
RANK() OVER ( [ <partition_by_clause> ] < order_by_clause > )
```

Syntax NTILE

```
NTILE (integer_expression) OVER ( [ <partition_by_clause> ]
< order_by_clause > )
```

Sample

```
SELECT A.AccountID, A.OpeningBalance
   , RANK() OVER (ORDER BY A.OpeningBalance) AS BalanceRank
   , DENSE_RANK() OVER (ORDER BY A.OpeningBalance)
      AS BalanceDenseRank
   , ROW_NUMBER() OVER (ORDER BY A.OpeningBalance)
      AS BalanceRowNumber
   , NTILE(10) OVER (ORDER BY A.OpeningBalance)
      AS BalanceGroupNumber
FROM Account AS A
;
```


Copyright © 2018 Ann L. Weber

	AccountID	OpeningBalance	BalanceRank	BalanceDenseRank	BalanceRowNumber	BalanceGroupNumber
1	3	0.00	1	1	1	1
2	11	0.00	1	1	2	1
3	15	0.00	1	1	3	1
49	255	0.00	1	1	4	2
50	263	0.00	1	1	50	2
51	274	0.00	1	1	51	2
52	135	10.00	52	2	52	2
53	110	10.00	52	2	53	2
54	105	10.00	52	2	54	2
55	95	10.00	52	2	55	2
56	35	10.00	52	2	56	2
57	241	320.53	57	3	57	3
58	68	662.28	58	4	58	3

Figure 77: Partial Results Set

Chapter 6 - Joining Multiple Tables

In this chapter:

JOINS
INNER JOIN
OUTER JOIN
CROSS JOIN
Joining Three or More Tables
Self-join
Alternate Syntax, Implicit Joins
Set operations
Viewing graphical execution plans
Chapter 6 Lab
Answers to Exercises

Files needed:

- \Chapter 06 JOIN\Try It Exercises
- \Chapter 06 JOIN\Labs\
- \Chapter 06 JOIN\Inline Samples
- \Student Files

	Some of the Try It exercises in this chapter build on one another, but are independent of other chapters. Answer files can be found in the \Chapter 06 JOIN\Try It Exercises folder.

 Copyright © 2018 Ann L. Weber

JOINS

In a relational database, you will often need to pull data from multiple tables within the same query. To accomplish this, you will need to use the JOIN clause. When you perform a join between two or more tables, you will include an ON clause to define what columns between the tables have the same meaning. This allow SQL to correlate the data in the different tables.

SQL includes a number of join types that allow to define what rows from each table are included in the result set.

In the ANSI-92 standard, all parts of the join definition are included in the FROM clause. The full FROM clause syntax is listed below.

Syntax

```
[FROM {<table_source>} [,...n]]

<table_source> ::=
{
      table_or_view_name [[AS] table_alias]
    | <joined_table>
}

<joined_table> ::=
{
    <table_source> <join_type> <table_source>
    ON <search_condition>
    | <table_source> CROSS JOIN <table_source>
}

<join_type> ::=
    [{INNER|{{LEFT|RIGHT|FULL}[OUTER]}}
```

INNER JOIN

Syntax

```
FROM Table1 [INNER] JOIN Table2
   ON <Search_Condition>
```

As you can see from the syntax, the word INNER is optional. When the query analyzer runs into the JOIN key word, it will perform an inner join unless additional key words are specified.

The search condition will typically be primary key column from table one being equal to foreign key column from table 2. This will not always be the case though. SQL allows the ON search condition to compare any two columns, regardless of the presence or absence of keys, as long as the data types are compatible. You can perform a join on two columns that are completely unrelated to on another. The data you get back will be useless, but SQL will run the query.

Copyright © 2018 Ann L. Weber

Additionally, you may find ON clauses that include multiple search conditions like when you have a composite key (two columns that make an individual row in a table unique), a combination of the two columns must frequently be used in the on clause to properly create the relationship. You will see a sample of this in the Try It exercise that follows this section. Additionally, in the case of self-joins, you may add additional conditions that do not use the equals sign at all.

	Although there are times when putting all search conditions in the ON clause rather than the WHERE clause can improve performance, this practice may produce very different result sets and can possibly slow performance as well. Be sure to know how making this change affects your result set and use the method that gives you the correct data. If the results sets never vary for your particular query, use the method that provides the best performance. Generally, the ON clause defines how the data is connected, and the WHERE clause limits the data after it is combined .

Sample

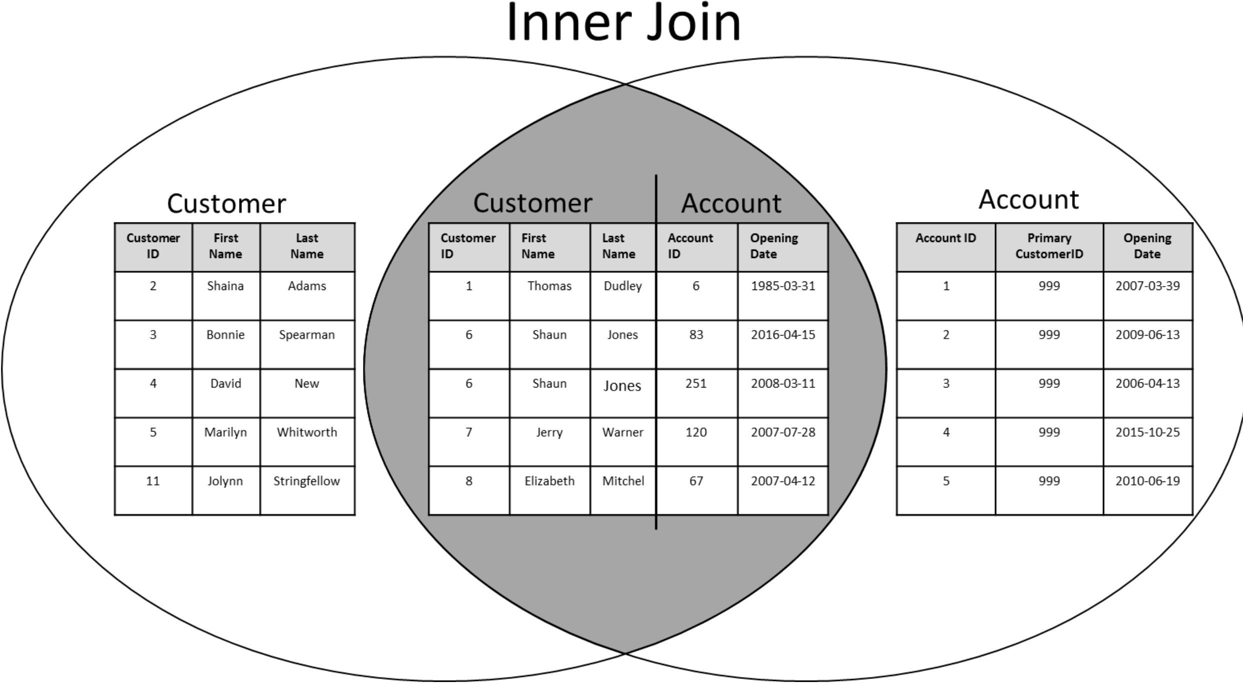
```
SELECT C.CustomerID, C.FirstName, C.LastName
  , CA.AccountID, CA.AccountNumber
FROM Customer AS C
  INNER JOIN CustomerAccount AS CA
  ON C.CustomerID = CA.CustomerID
;
```

In the sample above, the table alias names, “C” and “CA”, are optional. Table names (or table alias names) only needs to be referenced when the column name can be found in both tables, like the CustomerID column, making the column name ambiguous.

	Many programmers and organizations recommend (or require) putting the table name or alias in front of every column name in the query. This practice avoids the possibility of an ambiguous column error message in addition to making the query more readable. This way you can easily reference what columns are in which tables without having to reference the Object Explorer or external documentation.

With an INNER JOIN, only those rows with matching values specified in the ON clause are returned. For example, when joining the customer and account tables, an INNER JOIN will only return customers who exist in both tables where the PrimaryCustomerID field of the Account table matches the CustomerID column in the Customer table as shown in Figure 78.

 Copyright © 2018 Ann L. Weber



Customer ID	First Name	Last Name
2	Shaina	Adams
3	Bonnie	Spearman
4	David	New
5	Marilyn	Whitworth
11	Jolynn	Stringfellow

Customer ID	First Name	Last Name	Account ID	Opening Date
1	Thomas	Dudley	6	1985-03-31
6	Shaun	Jones	83	2016-04-15
6	Shaun	Jones	251	2008-03-11
7	Jerry	Warner	120	2007-07-28
8	Elizabeth	Mitchel	67	2007-04-12

Account ID	Primary CustomerID	Opening Date
1	999	2007-03-39
2	999	2009-06-13
3	999	2006-04-13
4	999	2015-10-25
5	999	2010-06-19

Figure 78: Inner Join

Try It 1 – INNER JOIN

In this exercise, you will use INNER JOIN statements to return information about primary customers and their account information. You will also find the AccountNumber associated with the primary customer of an account. The completed Try It Exercise scripts for each exercise can be found in the **Chapter 06 JOIN\Try It Exercises** folder.

1. Open a new query and click **File | Save** (or click the Save icon). Browse to the **\Student Files** folder, change the File name to **InnerJoin.sql**, and then click **Save**.
2. Use the RetailBanking database to write an inner join to return the **CustomerID**, **FirstName**, and **LastName** from the **Customer** table along with the **AccountID** and **OpeningDate** from the **Account** table. Only the customers with accounts should be returned. Type and execute the query below to achieve this goal. **280 rows** should be returned.

```
SELECT C.CustomerID, C.FirstName, C.LastName
     , A.AccountID, A.OpeningDate
FROM Customer AS C
     INNER JOIN Account AS A
     ON C.CustomerID = A.PrimaryCustomerID
;
```

3. Write and execute a query to return the AccountID, PrimaryCustomerID, and OpeningDate from the Account table similar to the one below. **280 rows** should be returned.

   ```
   SELECT A.AccountID, A.PrimaryCustomerID, A.OpeningDate
   FROM Account AS A
   ;
   ```

   Note: As previously mentioned, table aliases are not required, but make it easier to use IntelliSense and to add table references to the SELECT clause to improve readability.

4. Modify the query from step 3 to include the primary customer's account number for each of their accounts from the CustomerAccount table. Be careful, the combination of the CustomerID to PrimaryCustomerID and matching AccountID fields are both required to retrieve the correct matching row from the CustomerAccount table as shown below. **280 rows** should be returned.

   ```
   SELECT  A.AccountID, A.CreditLimit, A.OpeningDate
        , CA.AccountNumber
   FROM Account AS A
        INNER JOIN CustomerAccount AS CA
        ON A.PrimaryCustomerID = CA.CustomerID
             AND A.AccountID = CA.AccountID
   ;
   ```

5. Open Object Explorer and expand Databases | RetailBankingSample | to see the two gold colored keys for the composite primary key. The CustomerID and AccountID together create a composite key used to uniquely identify rows in the CustomerAccount table. The CustomerID key is to join the table back to the Customer table. Because accounts only hold the primary customer key, the AccountID key and the PrimaryCustomerID key together must be used to join to the Account table. SQL doesn't support direct many to many relationships, but an intermediary table with a composite or surrogate key can overcome that limitation.

6. Add both customer ID fields and both account id fields to the query with only one search condition to see that when the CustomerID to PrimaryCustomerID is used alone, your result set includes multiple rows for each customer with more than one account and it maps different account ids on the same row of the result

 Copyright © 2018 Ann L. Weber

set. When only the AccountID is used, all secondary account holders are also returned in addition to the primary account holders.

	AccountID	AccountID	CreditLimit	OpeningDate	AccountNumber	PrimaryCustomerID	CustomerID
1	1	1	NULL	2007-03-29	PMorga1	299	299
2	2	2	NULL	2009-06-13	ABritt2	109	109
3	3	3	30000.00	2006-04-13	DSmith3	39	39
4	4	4	NULL	2015-10-25	LLawso4	132	132
5	174	4	NULL	2007-10-24	LLawso4	132	132
6	5	5	NULL	2010-06-19	MPicou5	198	198
7	110	5	NULL	2015-10-13	MPicou5	198	198
8	6	6	NULL	1985-03-31	TDudle6	1	1
9	7	7	NULL	1988-02-02	GPayne7	229	229
10	149	7	NULL	1990-03-02	GPayne7	229	229
11	191	7	NULL	2017-04-28	GPayne7	229	229
12	8	8	NULL	2006-07-15	JGordo8	190	190

Figure 79: Partial Results Set

```
SELECT  A.AccountID, CA.AccountID, A.CreditLimit, A.OpeningDate
      , CA.AccountNumber, A.PrimaryCustomerID, CA.CustomerID
FROM Account AS A
      INNER JOIN CustomerAccount AS CA
      ON
            A.PrimaryCustomerID = CA.CustomerID
            --AND
            --A.AccountID = CA.AccountID
;
```

7. Save your query and leave SSMS open for the next Try It exercise.

Unless Query or JOIN hints are defined, the order of the order in which you list the tables in the JOIN clause is not important. Additionally, the order of the fields in the ON clause has no effect on the query results or performance as long as the condition is an equals (=) or not equals (<>) sign.

OUTER JOIN

In an OUTER JOIN, one or both tables will be considered the outer table. All rows will be returned from the outer table while only matching rows are returned from the inner table. For example, if the Customer table has customers who no longer have accounts or who do not yet have an account, using an outer join on the Customer and Account tables, with the Customer table as the “outer” table will return these customers.

Copyright © 2018 Ann L. Weber

Left and Right Outer Joins

In left and right outer joins, the outer table is determined based on its position in relation to the word join and the defining words of LEFT or RIGHT. Although this is less obvious when we add hard returns to make a query more readable, the table before the word join is considered to be on the left side of the join and the table name located after the word join is considered the right side.

Syntax

```
FROM Table1
   {{LEFT|RIGHT}[OUTER]}} JOIN
Table2
   ON <Search_Condition>
```

Samples

Sample 1

```
SELECT C.FirstName, C.LastName, A.AccountID
FROM Customer AS C
   LEFT OUTER JOIN  Account AS A
   ON C.CustomerID = A.PrimaryCustomerID
;
```

Sample 2

```
SELECT C.FirstName, C.LastName, A.AccountID
FROM Account AS A
   RIGHT OUTER JOIN  Customer AS C
   ON C.CustomerID = A.PrimaryCustomerID
;
```

Samples 1 and 2 above will return the same exact result set. This is because the Customer table is always the “outer” table. In these queries, the AccountID field will be NULL for those customers that do not have any current accounts listed in the database.

 Copyright © 2018 Ann L. Weber

Left Outer Join

Customer

Customer ID	First Name	Last Name
2	Shaina	Adams
3	Bonnie	Spearman
4	David	New
5	Marilyn	Whitworth
11	Jolynn	Stringfellow

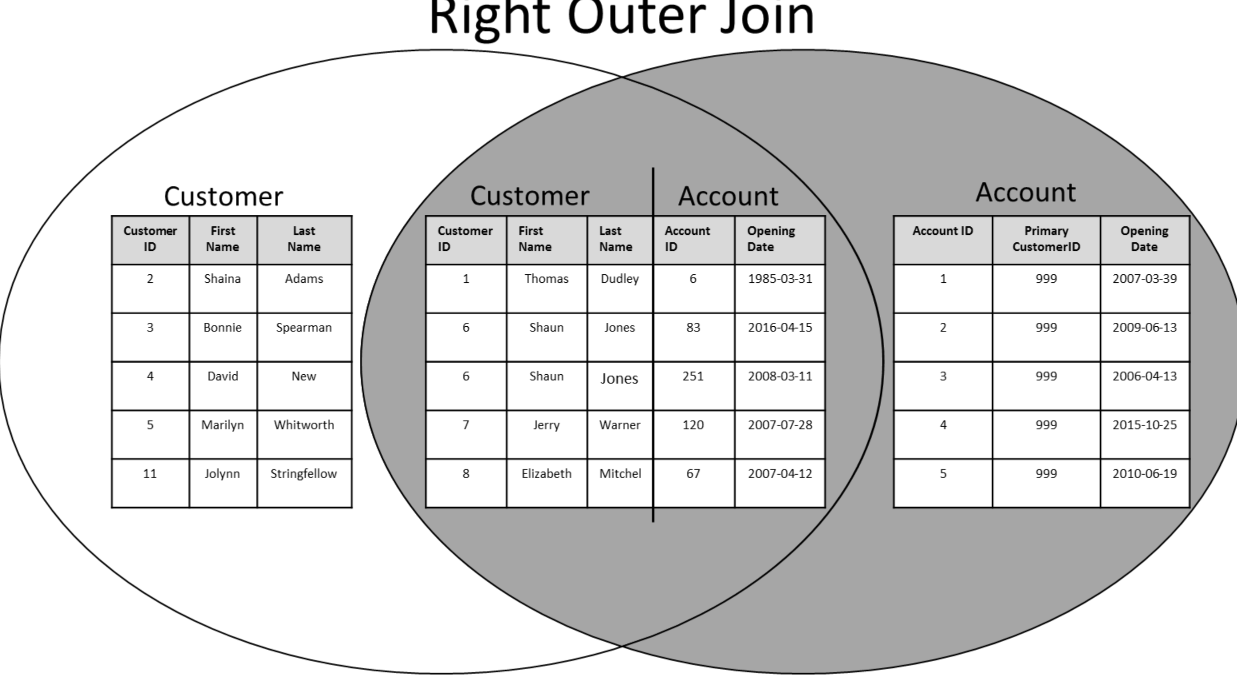
Customer / Account

Customer ID	First Name	Last Name	Account ID	Opening Date
1	Thomas	Dudley	6	1985-03-31
6	Shaun	Jones	83	2016-04-15
6	Shaun	Jones	251	2008-03-11
7	Jerry	Warner	120	2007-07-28
8	Elizabeth	Mitchel	67	2007-04-12

Account

Account ID	Primary CustomerID	Opening Date
1	999	2007-03-39
2	999	2009-06-13
3	999	2006-04-13
4	999	2015-10-25
5	999	2010-06-19

Figure 80: Left Outer Join



Customer ID	First Name	Last Name
2	Shaina	Adams
3	Bonnie	Spearman
4	David	New
5	Marilyn	Whitworth
11	Jolynn	Stringfellow

Customer ID	First Name	Last Name	Account ID	Opening Date
1	Thomas	Dudley	6	1985-03-31
6	Shaun	Jones	83	2016-04-15
6	Shaun	Jones	251	2008-03-11
7	Jerry	Warner	120	2007-07-28
8	Elizabeth	Mitchel	67	2007-04-12

Account ID	Primary CustomerID	Opening Date
1	999	2007-03-39
2	999	2009-06-13
3	999	2006-04-13
4	999	2015-10-25
5	999	2010-06-19

Figure 81: Right Outer Join

FULL OUTER JOIN

A full outer join treats both tables as outer tables and returns every row from each table, filling in any missing information with NULL values. If your database has foreign keys to protect all relationships, then a full outer join will return the same result set as the query where the table with the primary key is on the outer side of the join. This is because a foreign key protects against a row being added where the related primary key does not already exist.

Syntax

```
FROM Table1
   {FULL [OUTER]}} JOIN
Table2
   ON <Search_Condition>
```

Sample

```
SELECT C.CustomerID, C.ZipCode
   , CA.AccountID, CA.AccountNumber
FROM CustomerAccount AS CA
   FULL OUTER JOIN Customer AS C
   ON C.CustomerID = CA.CustomerID
;
```

A full outer join is very helpful in a database without foreign key constraints to help locate rows that would break the foreign key if one existed. For example, in the RetailBankingSample database, there are EmployeeID values in the Account table that would break a foreign key constraint if one existed between the Account and Employee tables. You will use a full outer join in the Try It below to locate both Employees who have never helped someone open an account, along with the invalid EmployeeID values in the Account table.

 Copyright © 2018 Ann L. Weber

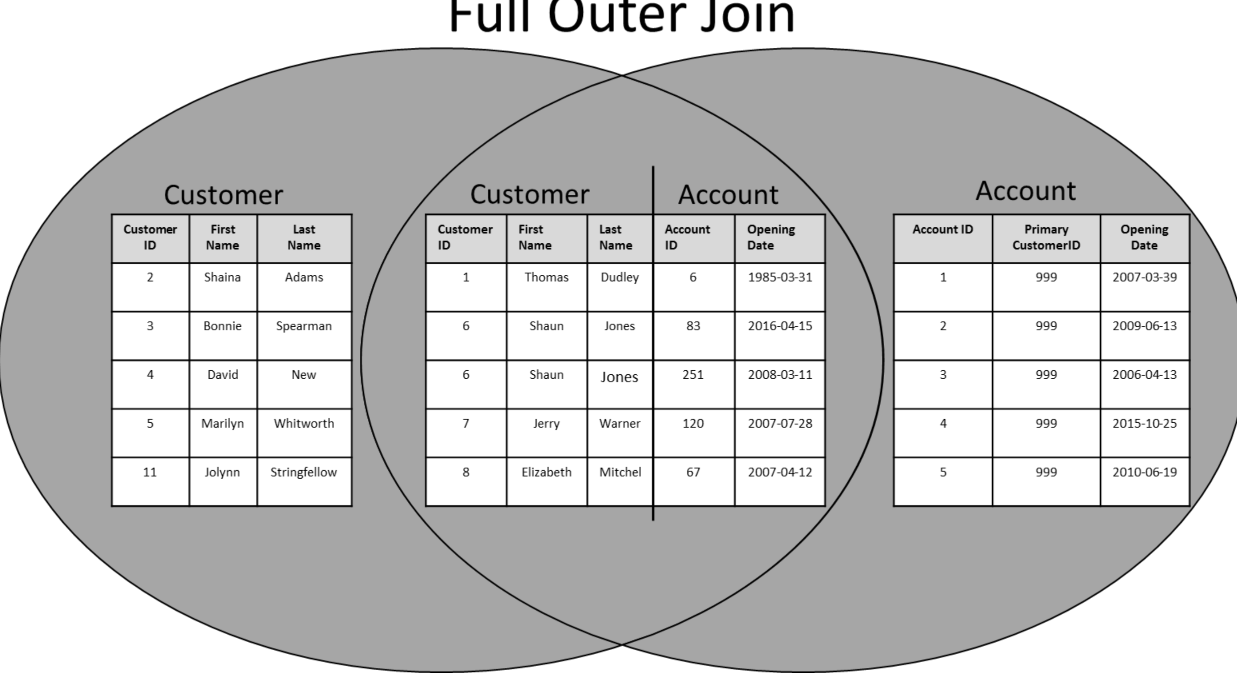


Figure 82: Full Outer Join

Try It 2 – Outer Joins

In this exercise, you will modify the inner join created in the first step of the previous Try It to return all customers along with the information about the customer's accounts. Additionally, you will write a new query to determine which accounts have an invalid EmployeeID and if there are any employees who do not have matching rows in the Account table.

1. Copy the query from Step 1 of the previous Try It to a new query window. Save the query file to the \Student Files folder and name the file **OuterJoins.sql**.
2. Execute the query. If necessary, set the current database to **RetailBankingSample**. The result set should include **280 rows.**

```
SELECT C.CustomerID, C.FirstName, C.LastName
     , A.AccountID, A.OpeningDate
FROM Customer AS C
     INNER JOIN Account AS A
     ON C.CustomerID = A.PrimaryCustomerID
;
```

3. Modify the query so that it returns all customers. Hint: The word left or right will be chosen based on the location of the Customer table within the query. If Customer is first, then modify the query to appear as below. **396 rows** should be returned.

```
SELECT C.CustomerID, C.FirstName, C.LastName
     , A.AccountID, A.OpeningDate
FROM Customer AS C
     LEFT OUTER JOIN Account AS A
     ON C.CustomerID = A.PrimaryCustomerID
;
```

4. Write a query to return the AccountID, EmployeeID, and AccountTypeID from the Account table. Also, return the EmployeeID, FirstName, and LastName fields from the Employee table. Start with a join that will only return employees that exists in both tables as shown below:

```
SELECT A.AccountID, A.EmployeeID, A.AccountTypeID
     , E.EmployeeID, E.FirstName, E.LastName
FROM Account AS A
     INNER JOIN Employee AS E
     ON A.EmployeeID = E.EmployeeID
;
```

5. You need to locate both rows in the Account table that have an invalid EmployeeID value and also any employees that are never listed in the Account table within a single query. Modify the query to return all rows from both tables as shown below:

```
SELECT A.AccountID, A.EmployeeID, A.AccountTypeID
     , E.EmployeeID, E.FirstName, E.LastName
FROM Account AS A
     FULL OUTER JOIN Employee AS E
     ON A.EmployeeID = E.EmployeeID
;
```

6. Modify the query to make it easier to locate any rows in either table that don't match an EmployeeID in the other table. **Hint**: The SQL Server fills any fields that it does not have information for with NULL values. See the query in the **Try It 2 – Outer Joins.sql** file in the **\Chapter 06\Try It Exercises** folder if you need help.

```
SELECT A.AccountID, A.EmployeeID, A.AccountTypeID
     , E.EmployeeID, E.FirstName, E.LastName
FROM Account AS A
     FULL OUTER JOIN Employee AS E
     ON A.EmployeeID = E.EmployeeID
     WHERE E.EmployeeID IS NULL
          OR A.EmployeeID IS NULL
;
```

Note: Every employee has at least one matching row in the account table, so this

 Copyright © 2018 Ann L. Weber

result set only includes the two rows from the Account table with no matching rows in the Employee table.

7. Save your queries and leave SSMS open for the next Try It.

CROSS JOIN

A cross join contains a full Cartesian product of the tables specified. This means that every row in the first table will be combined with every row of the second table. If your two tables each have 100 rows, the CROSS JOIN result set will include 10,000 rows. The performance hit from accidental cross joins is one of the reasons that using the ANSI-92 join syntax is recommended. Although there are some valid uses for performing a cross join, extreme care should be taken when making this decision.

Joining Three or More Tables

In a typical database, you will frequently need to join three or more tables to find all of the information that you need. For example, in our retail banking sample, if you want to display the primary customer's name, the initial loan amount, and the current loan balance, you will need at least three tables, namely Customer, Account, and LoanTransaction. Even if you do not desire to return any information in the Account table, this intermediary table must be referenced in the FROM/JOIN clause because there is no direct relationship between the Customer and LoanTransaction tables.

Syntax

```
FROM Table1
   [INNER][{{LEFT|RIGHT}[OUTER]}}] JOIN
Table2
   ON <Search_Condition>
   [INNER][{{LEFT|RIGHT}[OUTER]}}] JOIN
Table3
   ON <Search_Condition>
```

Sample

```
SELECT C.FirstName, C.LastName

   , LT.Amount, LT.TransactionDate, LT.TransactionType

FROM Customer AS C

   INNER JOIN Account AS A

   ON C.CustomerID = A.PrimaryCustomerID

   INNER JOIN LoanTransaction AS LT

   ON LT.AcctID = A.AccountID;
```

Try It 3 – Joining Three or More Tables

In this exercise you will open an existing query that combines information found in the Customer and Account tables. You will add the AccountTransaction table so that you can add the Amount, TransactionDate, and TransactionType columns to the result set. The result set should only include those customers with transactions in the AccountTransaction list. If time permits, you will also add in the Employee table to return the first and last name of the employee responsible for the new account when it was created.

1. Open the **Try It 3 – Joining More Tables Starter.sql** file in the **Chapter 06 JOIN\Try It Exercises** folder.
2. Click **File | Save Try It 3 – Joining More Tables Starter.sql As**, and then browse to the **\Student Files** folder. Type **Ch6TI3.sql** in the File name, and then click **Save**.
3. Modify the query to also include the **Amount**, **TransactionDate**, and **TransactionType** columns from the **AccountTransaction** table. If necessary, use the database diagrams to locate the fields that are related. Only those customers that have transactions in the **AccountTransaction** table should be included in the result set. If necessary, you can use the **Try It 3 – Joining More Tables Answer.sql** file. **81,174 rows** should be returned.
4. If time permits, also add in the **Employee** table to locate the employee name of the employees that was responsible for opening the account.
5. Save your query and close the query window. Leave SSMS open for the next Try It exercise.

Self-join

When you have a table where one column references another column in the same table, you will need a self-join to return the data based on this relationship. Since a self-join references the same table more than once, you must define an alias to track each "virtual" table reference. Using a meaningful alias will help you to determine the logic needed and make the query more readable.

In the Retail Banking Scenario, the Employee table includes a Foreign Key on the ReportsTo column that refers to the EmployeeID in the same table to locate the employee's manager. You will explore this relationship and write the self-join in the next Try It.

Another use of the self-join is when you need to locate groupings of items. For example, you want to return pairs of customers who are listed on the same account as shown in the sample below. The sample does not include accounts with only one associated customer.

Syntax

```
SELECT A.Col1, B.Col1 [, A.Col2, ...]
FROM Table1 AS A
  [INNER][{{LEFT|RIGHT}[OUTER]}}] JOIN
Table1 AS B
```

 Copyright © 2018 Ann L. Weber

```
ON <Search_Condition>
```

Sample

```
SELECT CA1.AccountID
   , CA1.CustomerID, CA2.CustomerID
FROM CustomerAccount AS CA1
   INNER JOIN CustomerAccount AS CA2
   ON CA1.AccountID = CA2.AccountID
        AND CA1.CustomerID < CA2.CustomerID
ORDER BY CA1.CustomerID, CA2.CustomerID
;
```

As you can see from the sample above, this join requires an additional search condition so that the result set does not include pairs with the same customer in both columns or the same pair, but in reverse order. Although a query without the additional search condition meets the criteria of pairs of customers on the same account, it does not meet the intended purpose of having unique pairs that you can use for analysis or for running a promotion.

Try It 4 – Self Join

In this exercise you will work through the process step by step of creating a self-join to better understand why and how of writing self-joins in the future.

1. Open a new query and click **File | Save** (or click the Save icon). Browse to the **\Student Files** folder, change the File name to **SelfJoin.sql**, and then click **Save**.
2. Write a query to return the **EmployeeID**, **FirstName and LastName** concatenated with a **space** between them as **EmployeeName**, **Title**, and **ReportTo** columns from the Employee table as shown below. If needed, you can copy the starter query from the **Try It 4 – Self-Joins.sql** file.

```
SELECT E.EmployeeID
     , E.FirstName + ' ' + E.LastName AS EmployeeName
     , E.Title, E.ReportsTo
FROM Employee AS E
;
```

3. Review who are the top level employees with no one to report to. How many rows are returned? Who is Howard Yeager's (Employeeid 4) manager? Think about the process that you used to determine that information. You will use this same process to set the search condition for the join.

4. Modify the query to create an inner join with the employee table. You will need to use an alias. Using E and M (for employee and manager) will help you track when you are referencing the Employee table to find employee info or manager info. Repeat the EmployeeID (temporarily for verification and troubleshooting only), the first and last names concatenated as ManagerName, and the title aliased as ManagerTitle as shown below. **48 rows** should be returned.

```
SELECT E.EmployeeID
     , E.FirstName + ' ' + E.LastName AS EmployeeName
     , E.Title, E.ReportsTo
     , M.EmployeeID
     , M.FirstName + ' ' + M.LastName AS ManagerName
     , M.Title
FROM Employee AS E
     INNER JOIN Employee AS M
     ON E.ReportsTo = M.EmployeeID
;
```

5. Review the query and use the E.EmployeeID , E.ReportsTo, and M.EmployeeID columns to verify that the search condition was entered correctly. If it is correct, comment out the M.EmployeeID column.
6. Who is missing from this result set and why?
7. Modify the query to include the missing managers. Think about whether it will be a left or right join. The query is shown below. **50 rows** should be returned.

```
SELECT E.EmployeeID
     , E.FirstName + ' ' + E.LastName AS EmployeeName
     , E.Title, E.ReportsTo
     --, M.EmployeeID
     , M.FirstName + ' ' + M.LastName AS ManagerName
     , M.Title
FROM Employee AS E
     LEFT OUTER JOIN Employee AS M
     ON E.ReportsTo = M.EmployeeID
;
```

8. Review the data.
9. Save and close your query, but leave SSMS open for the next Try It exercise.

	EmployeeID	EmployeeName	Title	ReportsTo	ManagerName	Title
1	1	William Diamond	Manager	NULL	NULL	NULL
2	2	Aurelio Bryant	Manager	NULL	NULL	NULL
3	3	Jill Moore	Banker	18	Jane Strickland	Supervisor
4	4	Howard Yeager	Banker	36	Andra White	Supervisor
5	5	Nancy Peery	Banker	37	Brittany Connors	Supervisor

Figure 83: Result Set

 Copyright © 2018 Ann L. Weber

Alternate Syntax, Implicit Joins

The ANSI SQL-89 specification can be found in a lot of historical code. These joins are defined implicitly by listing all of the tables in the FROM clause, and then defining the join search condition(s) in the WHERE clause. Microsoft has deprecated this syntax and recommends updating existing code to the syntax you learned up to this point in this chapter. The outer join ANSI SQL-89 syntax stopped working in SQL 2005.

Reasons to avoid ANSI SQL-89 syntax:

- You can accidentally perform a CROSS JOIN if you forget the WHERE clause.
- Most people find the ANSI-92 JOIN and ON syntax easier to read.
- Support for ANSI-89 outer join syntax *= and =* are not supported in compatibility modes 90 (SQL 2005) or later.

Set operations

So far in this chapter you have been working with JOINS, combining the data from multiple tables into a single result set. For three remainder of the chapter, we'll be learning about three commands that allow you to combine multiple result sets into a single result set.

UNION

The UNION statement takes the results of two SELECT statements and turns them into a single result set. You can merge more than two result sets by simply repeating the UNION operator additional times and adding more SELECT queries.

Syntax

```
select_query
UNION [ALL]
select_query
```

Sample

```
SELECT C.FirstName, C.LastName, C.ZipCode
FROM Customer AS C
UNION ALL
SELECT E.FirstName, E.LastName, 'Not available'
FROM Employee AS E
ORDER BY LastName, FirstName
;
```

The following rules and behaviors need to be considered when working with UNION:

1. Both SELECT statements must return the same number of columns. You can use NULL values, string literals, and concatenation among other methods to accomplish this.
2. Columns in the same position must return compatible data types. You can use CAST or CONVERT where implicit conversions are not sufficient.
3. Column alias definitions are only processed for the first query. Later alias definitions will be ignored, but will not cause errors.
4. Only one ORDER BY clause located after the final SELECT statement is supported.
5. The ORDER BY columns must exist in the SELECT statement.

By default, the SELECT statement returns all rows, including duplicates. If you want to remove duplicate rows, use SELECT DISTINCT. UNION is different in that it sorts the final result set and removes duplicates, unless it is modified by the keyword ALL.

	Use the ALL key word to improve performance of the query if you know that your two result sets will be returning unique rows. Additionally, you must include the ALL key word when you need to return the duplicate rows of which you might not be aware, such as with a financial system.

Try It 5 - Union

In this practice you will review the UNION operator by writing a query to return all of the transactions as well as a string literal explaining which table the transaction is coming from.

1. Open the **Try It 5 – Union Starter.sql** file from the **\Chapter 06 JOIN\Try It Exercises** folder.
2. Click **File | Save Try It 5 – Union Starter.sql As**, and then browse to the **\Student Files** folder. Type **UNION.sql** in the File name, and then click Save.
3. Modify the individual queries to match the requirements of the UNION statement.
4. Add a string literal to include the table name in which the rows reside.
5. Combine the result sets into a single result set that is optimized for performance.
6. Modify the query to sort the data from highest amount to lowest. If necessary, a sample of one possible final query can be found in **Try It 5 – Union Answer.sql** file from the **\Chapter 06 JOIN\Try It Exercises** folder.
7. Save and close your query, but leave SSMS open for the next Try It.

 Copyright © 2018 Ann L. Weber

INTERSECT

The INTERSECT operator combines two result sets and returns a result set containing distinct rows that exist in both result sets. For example, you could use INTERSECT to return the combination of cities and states where both customers and employees live or to find out which accounts had transactions in two separate years.

Like the UNION operator, both select statements included with INTERSECT need to have the same number of columns with compatible data types and column aliases defined in the first query. If the ORDER BY clause if used, it must be at the very end of the combined statements and include only columns listed in the SELECT statements.

Like UNION, you can continue this process by including additional SELECT statements with the INTERSECT operator between them.

Syntax

```
select_query
INTERSECT
select_query
```

Sample

```
SELECT CT.AcctID
FROM CreditTransaction AS CT
WHERE TransactionDate BETWEEN '20170101' AND '20171231'
INTERSECT
SELECT CT.AcctID
FROM CreditTransaction AS CT
WHERE TransactionDate BETWEEN '20160101' AND '20161231'
INTERSECT
SELECT CT.AcctID
FROM CreditTransaction AS CT
WHERE TransactionDate BETWEEN '20150101' AND '20151231'
ORDER BY AcctID
;
```

Try It 6 - Intersect

In this exercise you will write a query implementing the INTERSECT operator. If needed, the answer can be found in the **Try It 6 – Intersect.sql** file located in the **\Chapter 06 JOIN\Try It Exercises** folder.

1. Open a new query and click **File | Save** (or click the Save icon). Browse to the **\Student Files** folder, change the File name to **Intersect.sql**, and then click **Save**.
2. Using the Sample above as your guide, write a query that returns the account id and account transaction type for account transactions where the AccountID had ATM transactions in both 2016 and 2017. The ATM value is located in the TransactionType column.
3. If time permits, extend your query to include only accounts with ATM transactions in the years 2015, 2016, and 2017.
4. Save your query and leave it open for the next Try It.

EXCEPT

The EXCEPT operator takes two result sets and returns all rows from the first set that are NOT included in the second result set. You can use EXCEPT to find accounts with transactions this year that did not also have transactions last year. Unlike UNION and INTERSECT where the order of the SELECT statements doesn't matter, changing the order with SELECT statements with the EXCEPT operator changes the meaning and the ensuing result set.

Syntax

```
select_query
EXCEPT
select_query
```

Sample

```
SELECT CT.AcctID
FROM CreditTransaction AS CT
WHERE TransactionDate BETWEEN '20170101' AND '20171231'
EXCEPT
SELECT CT.AcctID
FROM CreditTransaction AS CT
WHERE TransactionDate BETWEEN '20160101' AND '20161231'
;
```

Try It 7 - Except

In this exercise you will write a query implementing the intersect operator. If needed, the answer can be found in the **Try It 7 – Except.sql** file located in the **\Chapter 06 JOIN\Try It Exercises** folder.

1. Modify your query from the previous Try It exercise to show any accounts that only had ATM transactions in 2017, but not in 2016. Remember, the order of your queries now matters.

 Copyright © 2018 Ann L. Weber

Note: If you did not complete Try It 6 you can open Try It 6 - Intersect.sql and use the Save … As option to save a copy to the \Student Files folder.

2. Save your query.

Viewing graphical execution plans

Frequently when writing joins, you need to analyze the performance of your queries. Although a full discussion of execution plans is beyond the scope of this course, you can quickly see the performance difference between two different ways of writing the same query by viewing either the estimated or actual execution plans. You can enable graphical execution plans with the icons in the SQL Editor toolbar, or by using the Query menu.

	There are some great inexpensive or free books available on understanding the execution plan including, SQL Server Execution Plans by Grant Fritchey at http://download.red-gate.com/ebooks/SQL/sql-server-execution-plans.pdf

Try It 8 – Execution Plans

In this Try It, your instructor will guide you through comparing two queries that will return the same data, but will perform differently.

1. Open the **Try It 8 – Execution Plans Starter.sql** file from the **\Chapter 06 JOIN\Try It Exercises** folder.
2. Use the **File | Save … As** menu option to save the query to your \Student Files folder.
3. Execute the queries under Step #2 to enable statistics on both disk/memory i/o and on CPU usage.
4. Click the Include Actual Execution plan icon or press **Ctrl + M**.
5. Execute the query under step #4.
6. Switch to the Execution plan tab and notice the green Missing Index comment.
7. Right-click on the Missing Index comment, and then click **Missing Index Details**. Review the table and columns that are suggested for inclusion in the index. Do not create the index at this time, but leave the tab open for later.
   Note: You will learn more about Indexes in **Chapter 10: Data Definition Language**
8. Adding WHERE clauses to a query will help performance, especially when the correct indexes exist. Highlight and execute the queries under Both Step #4 and Step #7 at the same time. Notice the estimated **Query cost (relative to the batch):** values for each query. Even without indexes to support the queries, the query

with the where clause is significantly more expensive, even though the plans are currently the same.

9. Change to the Messages tab. Scroll down to the (91756 row(s) affected) note. Each logical read represents an 8k page in memory. You will only have physical reads if the data is not currently in cache. For every physical read, there will also be a logical read. Notice that currently, both queries have the same IO stats, but depending on the circumstances, you may see a difference in the CPU time under the SQL Server Execution Times that equate to each query.

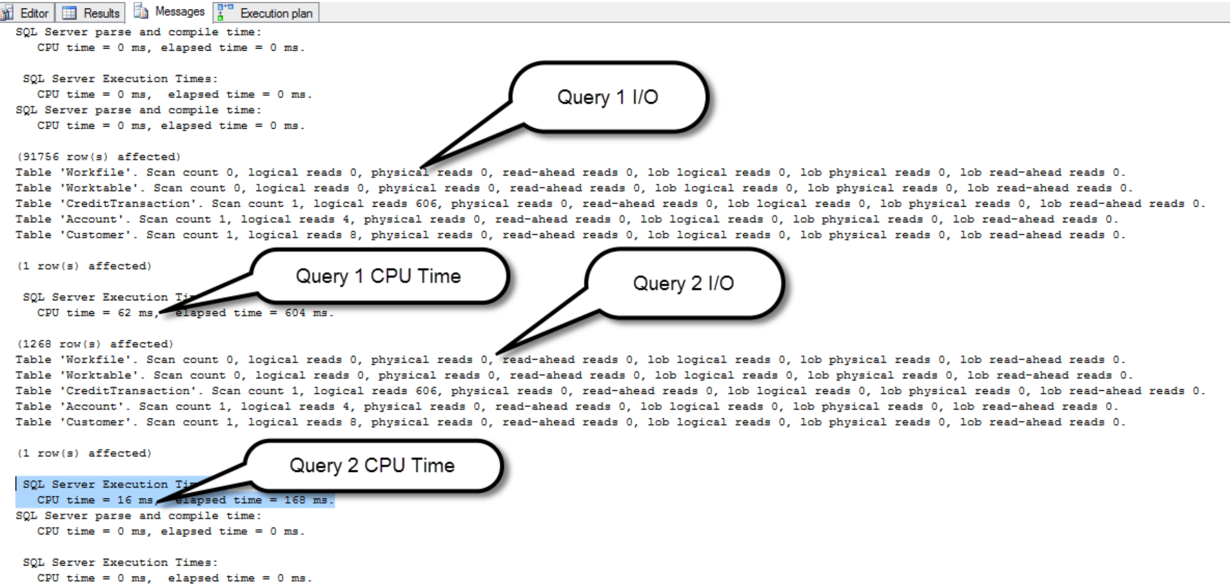


Figure 84: Messages tab

10. Return to the Execution plan tab. Right-click on the Missing Index hint for **Query 2**, and click **Missing Index Details**.
11. Remove the <Name of Missing Index, sysname, > from Line 9 of the query window and replace it with nc_CreditTransaction_TransactionDate. The query should now look like the query below.

```
USE [RetailBankingSample]
GO
CREATE NONCLUSTERED INDEX
[nc_CreditTransactionTransactionDate]
ON [dbo].[CreditTransaction] ([TransactionDate])
INCLUDE ([CreditTrxID],[AcctID],[Amount])
GO
```

12. Highlight the code inside of the block comments or remove the second set of block comments and execute the code.
13. In the Object Explorer, expand the **RetailBankingSample | Tables | dbo.CreditTransaction | Indexes** folders. If necessary, right-click the **Indexes**

 Copyright © 2018 Ann L. Weber

folder and click **Refresh** if you don't see the nc_CreditTransaction_TransactionDate index.

14. Return to your query window from step 7. If you can't locate the query window, or if you closed it, rerun the query under Step #4, and then right-click the Missing Index hint and click Missing Index Details.
15. Change the name of this index to **nc_CreditTransaction_AcctID** and then execute the commands inside of the second set of block comments.
16. Right-click the **Indexes** folder under the **dbo.CreditTransaction** table, and then click **Refresh** to verify the creation of the second index.
17. Rerun the Step #4 and Step #7 queries once again at the same time.
18. Review the Execution plan and Messages tabs. Notice that with the new indexes, the performance difference is even greater. Both queries now perform fewer reads on the CreditTransaction table.
19. Execute the queries under – - Step 18 Cleanup.
20. Save your changes and close the query window.

Chapter 7 - Subqueries

In this chapter:

Subqueries
Nested vs Correlated Subqueries
Subqueries in the SELECT Clause
Subqueries in the WHERE Clause
EXISTS
Subqueries in FROM Clause
Alternatives to Subqueries
Chapter 7 Lab
Answers to Exercises

Files needed:

- \Chapter 07 Subqueries\Inline Samples
- \Chapter 07 Subqueries\Try It Exercises
- \Chapter 07 Subqueries\Labs\

	Try It answer files can be found in the \ Chapter 07 Subqueries\Try It Exercises folder.

 Copyright © 2018 Ann L. Weber

Subqueries

Simply put, a subquery is a query within a query. Subqueries are frequently used to overcome limitations with SQL syntax rules or to break complex queries into more manageable pieces. Subqueries can be used in the SELECT, FROM, HAVING, and WHERE clauses of a SELECT statement. Subqueries can also be used with INSERT, UPDATE, and DELETE statements.

	You will learn more about the INSERT, UDPATE, and DELETE Data Manipulation Language (DML) statements in **Chapter 9 Data Manipulation Language**.

Below is a list of some of the subquery rules that may apply depending on the location of the subquery.

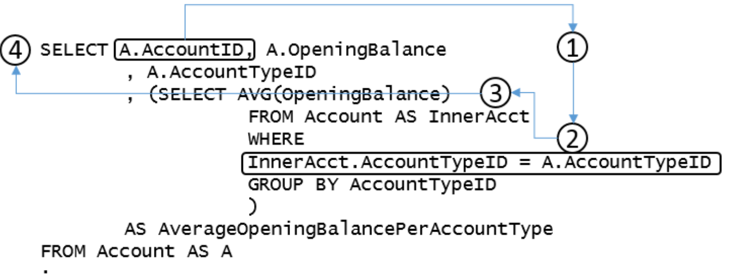
- The SELECT statement of the subquery is always enclosed in parentheses.
- An ORDER BY clause cannot be used in a subquery unless the TOP clause is also specified.
- Nesting of any type cannot exceed 32 levels.
- Tables only included in the subquery cannot be referenced in the outer query SELECT list.
- Ntext, text, and image data types cannot be included in the SELECT list of the subquery.
- The DISTINCT key word cannot be used in subqueries that also use GROUP BY.

Nested vs Correlated Subqueries

Subqueries can be either simply nested within another query – referred to as nested subqueries, or correlated and linked row by row to the outer query – referred to as correlated subqueries. The easiest way to recognize a nested subquery is by trying to execute the subquery by itself. If you leave off the parentheses, a nested subquery can be executed independently.

Correlated subqueries can be recognized by the inclusion of a search condition similar to what you see in the ON clause of a join. This search condition defines the correlation between the inner and outer queries.

With a correlated subquery, the server performs the following steps as shown in Figure 85.



1. Retrieves the outer row
2. Checks the value from the outer row against the search condition in the correlated subquery
3. Continues to process the row and return the data if the search condition returns "True"
4. Returns to the outer query and retrieves the next row, repeating the process for each row in the outer table.

Figure 85: Correlated Subquery Steps

Sample Nested Subquery

The following sample returns the average opening balance for all accounts with an account type of 12, along with the opening balance of the account on each row. This value could then be used to find the difference between that account's balance and the average or any other required formula. The inner select can be executed by itself, making it a simple nested subquery.

```
SELECT A.AccountID, A.OpeningBalance
   , (SELECT AVG(OpeningBalance)
         FROM Account WHERE AccountTypeID = 12)
   AS [AvgAllAcctType12OpeningBalance]
FROM Account AS A
WHERE AccountTypeID = 12
;
```

	AccountID	OpeningBalance	AverageAllAcountType12OpeningBalance
1	9	32963.78	29993.7594
2	13	58458.02	29993.7594
3	28	19676.97	29993.7594
4	37	9929.32	29993.7594
5	53	29528.74	29993.7594
6	76	28532.46	29993.7594
7	114	17622.02	29993.7594
8	157	41358.49	29993.7594

Figure 86: Partial Results Set

The above code could be rewritten as a correlated subquery to display the average opening balance per account type, rather than for a single account type. The difficult part is determining which field is needed to provide the correlation between the inner and outer query. In this case, the account type allows us to provide the average for the account type in the row being returned. Notice that the inner query cannot be run by itself. The inner query relies on the Account table aliased as A in the outer query.

Sample Correlated Subquery

```
SELECT A.AccountID, A.OpeningBalance, A.AccountTypeID
   , (SELECT AVG(OpeningBalance)
         FROM Account AS InnerAcct
```

 Copyright © 2018 Ann L. Weber

```
            WHERE InnerAcct.AccountTypeID = A.AccountTypeID
            GROUP BY AccountTypeID
        )
        AS [AvgOpeningBalancePerAcctType]
    FROM Account AS A
    ;
```

	AccountID	OpeningBalance	AccountTypeID	AverageOpeningBalancePerAccountType
1	1	33009.37	9	31767.4457
2	2	39800.27	2	27342.8021
3	3	0.00	13	0.00
4	4	37048.13	5	25745.2611
5	5	17185.94	8	24653.051
6	6	34732.16	7	32416.8855
7	7	44147.88	5	25745.2611
8	8	43654.10	7	32416.8855
9	9	32963.78	12	29993.7594

Figure 87: Partial Results Set

Subqueries in the SELECT Clause

In the SELECT clause, the subquery returns one value per result set row displayed as a new column. As a best practice, alias the column containing the subquery. Otherwise, the result set will return **(No column name)**. Both of the prior samples demonstrated a subquery in the SELECT clause of the query. The sample below takes the nested subquery a step further by calculating the difference between the opening balance for the current row and the average opening balance for all accounts with an account type of 12.

Sample

```
SELECT A.AccountID, A.OpeningBalance
     , (SELECT AVG(OpeningBalance)
            FROM Account WHERE AccountTypeID = 12

       ) - A.OpeningBalance
       AS DifferenceFromAvg
FROM Account AS A
WHERE AccountTypeID = 12
;
```

If the formatting of the opening balance and average columns is bothering you, you can use the CONVERT function with a style of 1 to include a comma and display both columns with only 2 decimal places. The Inline Sample 07.sql file in the \Chapter 07 Subqueries\Inline Samples folder includes the CONVERT function.

Try It 1 – Subqueries in the SELECT

In this exercise, you will write a query to include the number of transactions in the LoanTransaction table per account. The query also needs to return the AccountOpeningDate and the OpeningBalance from the Account table for accounts with AccountTypeIDs of 6, 7, 8, and 9. A partial result set is shown in Figure 88. AccountTypeIDs 6-9 represent mortgage account types, which are the account types linked to the LoanTransaction table.

	AccountID	OpeningDate	OpeningBalance	NumberTransactions
1	1	2007-03-29	33009.37	267
2	5	2010-06-19	17185.94	189
3	6	1985-03-31	34732.16	359
4	8	2006-07-15	43654.10	283
5	16	2011-11-22	44134.68	155
6	18	2010-02-05	43787.51	197
7	27	1988-04-23	41924.68	719
8	30	2005-04-06	34522.29	313

Figure 88: Partial Result Set

1. Open a query window, save the query as \Student Files\SubquerySELECT.sql, and set the current database to RetailBankingSample.
2. Write and execute a query to return the AccountID, OpeningDate, and OpeningBalance fields from the Account table for account with an AccountID between 6 and 9 as shown below. Your query should return **86 rows**.

```
SELECT A.AccountID, A.OpeningDate, A.OpeningBalance
FROM Account AS A
WHERE AccountTypeID BETWEEN 6 AND 9
;
```

3. Write and execute a query that returns the count of all rows in the LoanTransaction table.

```
SELECT COUNT(*)
FROM LoanTransaction AS LT
;
```

 Copyright © 2018 Ann L. Weber

4. Take the two queries that you just wrote and put them together to make a single query that returns the result set described in the introduction to this Try It. You will need to add a WHERE clause to the inner query to correlate it to the outer query. A sample query is included below:

```
SELECT A.AccountID, A.OpeningDate, A.OpeningBalance
     , (SELECT COUNT(*)
          FROM LoanTransaction AS LT
          WHERE LT.AcctID = A.AccountID
       ) AS NumberTransactions
FROM Account AS A
WHERE AccountTypeID BETWEEN 6 AND 9
;
```

5. Save your query and close the current query window. Leave SSMS open for the next Try It exercise.

Subqueries in the WHERE Clause

A common use of subqueries is to limit the rows in a result set based on data that is returned from a different query. For example, the following query returns all account transactions from accounts associated with more than one customer.

Sample

```
SELECT AT.AcctID, AT.TransactionType, AT.Amount
FROM AccountTransaction AS AT
WHERE AT.AcctID IN (SELECT AccountID
   FROM CustomerAccount
   GROUP BY AccountID
   HAVING COUNT(*) > 1)
;
```

Some obvious and less obvious rules when working with subqueries in the WHERE clause include:

- Unless using EXISTS, the subquery must return a single column.
- If a comparison operator such as equal, less than, etc. is used in the WHERE clause of the outer query, a single value must be returned from the inner query. An exception to that rule is if the ANY or ALL keywords are also being used in the WHERE clause of the outer query. In these cases, multiple values can be returned from the inner query.
- With the IN keyword, multiple values in a single column are allowed.
- When using IN or a comparison operator, the column in the WHERE clause of the outer query must be of a compatible data type to the column in the SELECT list of the inner query.

Try It 2 – Subqueries in WHERE

In this exercise, you will practice adding a nested subquery to a where clause to return all account transactions placed on the most recent date in the database. Then, you will modify that query to use a correlated subquery that returns all of the transactions that were on the last transaction date for that account. Both the nested and correlated partial result sets can be seen in Figure 89 and Figure 90.

	AcctID	TransactionDate	Amount
1	96	2018-06-01 00:00:00	2289.42
2	174	2018-06-01 00:00:00	3653.71
3	35	2018-06-01 00:00:00	-1325.45
4	191	2018-06-01 00:00:00	823.74
5	222	2018-06-01 00:00:00	3396.79
6	269	2018-06-01 00:00:00	-509.34
7	20	2018-06-01 00:00:00	2244.99
8	56	2018-06-01 00:00:00	-621.62

Figure 89: Nested Query Results

	AcctID	TransactionDate	Amount
1	2	2018-05-25 00:00:00	-1431.87
2	4	2018-06-01 00:00:00	-1486.97
3	7	2018-05-28 00:00:00	2071.19
4	10	2018-05-28 00:00:00	-574.16
5	12	2018-05-28 00:00:00	-1135.97
6	17	2018-05-26 00:00:00	-2322.85
7	19	2018-05-27 00:00:00	1055.10
8	20	2018-06-01 00:00:00	2244.99

Figure 90: Correlated Query Results

Because of the data in this database, there is at most one transaction per day for each account. Therefore, we only get the final transaction for that account returned.

1. Open a query window, save the query as \Student Files\SubqueryWHERE.sql, and set the current database to RetailBankingSample.
2. Write a query as described below. If needed the full query is included below.
    a. Return the most recent date for any transaction in the AccountTransaction table, no matter what account, date, etc.
    b. Do not create an alias for the column name.
    c. Add InnerAT as an alias for the AccountTransaction table.
    d. A single value should be returned.

 Copyright © 2018 Ann L. Weber

e. Do not include a semi-colon. This query will later be used as a subquery.

```
SELECT MAX(InnerAT.TransactionDate)
FROM AccountTransaction AS InnerAT
```

3. Execute and test your query. **2018-06-01 00:00:00** should be returned.
4. Write a query to return the AcctID, TransactionDate, and Amount columns from the AccountTransaction table. If needed, the query is shown in the **Try It 2 – Subqueries in WHERE.sql** file.
5. Add a WHERE clause to the query you wrote in Step 4. The subquery from Step 2 should be used as the right-hand side of the comparison. Only rows where the TransactionDate is equal to the most recent date for any transaction in the table should be returned. Make sure that your semi-colon is AFTER the full WHERE clause. The finished query should look similar to the following query. The result set should return **25 rows** and resemble the partial result set shown in Figure 89 above.

```
SELECT AT.AcctID, AT.TransactionDate, AT.Amount
FROM AccountTransaction AS AT
WHERE AT.TransactionDate
           = (SELECT MAX(InnerAT.TransactionDate)
               FROM AccountTransaction AS InnerAT
              )
;
```

6. Modify the query in Step 5 to correlate the queries and return the transaction(s) for each account that occurred on the latest transaction date for that specific account. Order the results by AcctID as shown in the query below:

```
SELECT AT.AcctID, AT.TransactionDate, AT.Amount
FROM AccountTransaction AS AT
WHERE AT.TransactionDate = (SELECT
                                 MAX(InnerAT.TransactionDate)
                            FROM AccountTransaction
                                 AS InnerAT
                            WHERE InnerAT.AcctID
                                 = AT.AcctID
                            )
ORDER BY AT.AcctID
;
```

7. Save your query and close the current query window.

EXISTS

The EXISTS operator is used to check if any rows are returned by the subquery. It only returns a TRUE of FALSE indication for each row in the outer query. EXISTS can be used as a different way to write correlated subqueries.

Syntax

```
EXISTS (subquery)
```

Sample

```
SELECT C.CustomerID, C.FirstName, C.LastName
FROM Customer AS C
WHERE NOT EXISTS (SELECT 1 FROM Account AS A
                  WHERE A.PrimaryCustomerID = C.CustomerID)
;
```

Note: Additional Samples of queries with EXIST clauses and alternate query styles can be found at **\Chapter 07 Subqueries\Inline Samples\Additional EXISTS Samples.sql**. In the first example, all three queries produce the exact same execution plan. In the second example, the JOIN produces a different execution plan, but the actual CPU time and number of reads are identical in all three queries. The estimated cost of the JOIN is 1% higher than the other two, but the three costs need to add up to 100% and partial percentages are not displayed.

	Some sources suggest that using EXIST can lead to performance gains over other ways of writing identical queries, but such gains are highly dependent of the specifics of the particular query. If performance is important, you should test your queries using actual data in real world scenarios. With the more current versions of SQL Server, the query optimizer often returns the same query plan regardless of whether an EXISTS clause, other subquery, or a JOIN is used.

Subqueries in FROM Clause

When you use a subquery in the FROM clause, it is referred to as a derived table. This type of subquery is frequently used when you need to manipulate the data either by changing the column definitions, joining multiple tables together, performing aggregations, or other data manipulation before retrieving data based on the derived result set, rather than the original data.

There are other features you can also leverage when the need arises to perform complex queries. Some of these features include temp tables, table variables, and Common Table Expressions (CTEs). You will learn more about these additional features in **Chapter 11 Working with Temporary Objects**.

	To improve code readability, give your derived tables meaningful aliases. Additionally, avoid nesting multiple derived tables whenever possible.

The following sample will return account transactions for accounts with more than one linked customer. This query returns the same result set as the sample above but uses a derived table instead of a WHERE

 Copyright © 2018 Ann L. Weber

clause. Although the two samples look quite different, SQL uses the same execution plan for both queries.

Sample

```
SELECT AccountID, AT.TransactionType, AT.Amount
FROM (SELECT AccountID, COUNT(*) AS CustomerCount
  FROM CustomerAccount
  GROUP BY AccountID
  HAVING COUNT(*) > 1) AS MultiuserAccounts
INNER JOIN AccountTransaction AS AT
ON AT.AcctID = MultiuserAccounts.AccountID
;
```

Try It 3 – Subqueries in FROM

In this exercise, you will write a query that will use both table joins and a subquery in the FROM clause to display the customer ID, count of the number of accounts listed, the first and last names of the customer, the account id, and account type for customers who have more than one account in the CustomerAccount table.

A partial result set is shown in Figure 91. Although there are multiple ways to approach this query, this Try It exercise will walk you through the steps of one of these approaches.

	CustomerID	NumberofAccounts	FirstName	LastName	AccountID	AccountType
1	2	2	Shaina	Adams	1	30 Year ARM
2	3	2	Bonnie	Spearman	2	Simple Savings
3	2	2	Shaina	Adams	3	CreditCard
4	60	2	Minnie	Cayton	3	CreditCard
5	260	2	Darwin	Humphrey	3	CreditCard
6	4	2	David	New	4	Premium Savings
7	132	2	Lawrence	Lawson	4	Premium Savings
8	5	2	Marilyn	Whitworth	5	15 Year ARM

Figure 91: Partial Result Set

1. Open a query window, save the query as \Student Files\SubqueryFROM.sql, and set the current database to RetailBankingSample.
2. First, walk through the subquery that will become the basis for finding out which customers have more than one account. Because we want to see additional fields such as the account type, we can’t simply use an aggregate in the SELECT and a HAVING clauses. Start by writing a query that will return the CustomerID and the count of the AccountID field for all customers that have more than one account in the CustomerAccount table. **133 rows** will be returned. If necessary, use the code below:

Copyright © 2018 Ann L. Weber

```
SELECT CA.CustomerID
            , COUNT(CA.AccountID) AS NumberofAccounts
       FROM CustomerAccount AS CA
       GROUP BY CA.CustomerID
       HAVING Count(AccountID) > 1
```

Note: Because this is going to be used later as a subquery, you can leave the semi colon off of the end of the query.

3. Execute your query and note how many rows are returned (133 rows). Also note the first few CustomerID values so that you can see if any of them are lost when adding additional parts of the query.
4. Next, write a query joining multiple tables together to return the following columns from the tables listed. All rows in common between the tables should be returned. A sample query follows the fields list.
   Hint: Look at your database diagrams. A non-listed table is needed to locate the AccountType wording.
   a. **Customer** table
      i. CustomerID
      ii. FirstName
      iii. LastName
   b. **CustomerAccount** table
      i. AccountID
   c. **AccountType** table
      i. AccountType

```
SELECT C.CustomerID, C.FirstName, C.LastName
     , A.AccountID ,AT.AccountType
FROM Customer AS C
     INNER JOIN CustomerAccount AS CA
          ON C.CustomerID = CA.CustomerID
     INNER JOIN Account AS A
          ON A.AccountID = CA.AccountID
     INNER JOIN AccountType AS AT
          ON A.AccountTypeID = AT.AccountTypeID
;
```

5. Execute your query. **459 rows** should be returned.
6. Add an ORDER BY clause and sort by the **CustomerID**. Notice that CustomerID 1 only has one account, CustomerID 2 has two accounts, and CustomerID 12 has six accounts.

```
SELECT C.CustomerID, C.FirstName, C.LastName
     , A.AccountID ,AT.AccountType
FROM Customer AS C
     INNER JOIN CustomerAccount AS CA
          ON C.CustomerID = CA.CustomerID
     INNER JOIN Account AS A
          ON A.AccountID = CA.AccountID
     INNER JOIN AccountType AS AT
          ON A.AccountTypeID = AT.AccountTypeID
```

 Copyright © 2018 Ann L. Weber

```
ORDER BY CustomerID
;
```

7. Modify the query in Step 5 to include the subquery from step 1 in the FROM clause. Give the subquery an alias of Sub and join it to the other tables already in your outer query. The resulting rowset will include **298 rows** and will look similar to Figure 91 at the beginning of this Try It exercise. The final query is below if you need help.

```
SELECT C.CustomerID, Sub.NumberofAccounts
     , C.FirstName, C.LastName
     , A.AccountID ,AT.AccountType
FROM
     (SELECT CA.CustomerID
           , COUNT(CA.AccountID) AS NumberofAccounts
       FROM CustomerAccount AS CA
       GROUP BY CA.CustomerID
       HAVING Count(AccountID) > 1) AS Sub
     INNER JOIN Customer AS C
          ON Sub.CustomerID =C.CustomerID
     INNER JOIN CustomerAccount AS CA
          ON Sub.CustomerID = CA.CustomerID
     INNER JOIN Account AS A
          ON A.AccountID = CA.AccountID
     INNER JOIN AccountType AS AT
          ON A.AccountTypeID = AT.AccountTypeID
ORDER BY CA.CustomerID
;
```

Note: The final result set has more rows than the subquery but fewer rows than the original outer query. Although fewer rows are returned from the query in Steps 4 and 5 by eliminating all customers with only one account, we are increasing the number of rows from the subquery by joining in information for each account represented by the NumberofAccounts column. Thus, the final result set contains two rows for CustomerID 2 and six rows for CustomerID 12, and CustomerID 1 is eliminated.

8. If time permits, change the ORDER BY clause to sort by the AccountID. This will allow you to see the Many-to-many relationship being supported by the CustomerAccount intermediary table. Each customer can have multiple accounts and each account can be linked to multiple customers.
9. Save and close the query tab. Leave SSMS open for the next Try It exercise.

Alternatives to Subqueries

Many subqueries can be rewritten using GROUP BY and JOIN statements, but the performance often ends up being the same due to the query optimizer using the same execution plan. On the other hand, there are times the join will perform better than the subquery (frequently with correlated subqueries)

Copyright © 2018 Ann L. Weber

where the optimizer feels it needs to perform the analysis row by row. There are other times when the subquery will also perform better than the join.

<table>
<tr>
<td>
</td>
<td>In the past, many SQL experts recommended replacing subqueries with JOIN where ever possible. Since that time, the SQL Query Optimizer has improved rendering these replacements less important than before. If your organization still recommends replacing subqueries with joins, test both your data results and the performance. In some cases, a subquery performs more like an OUTER JOIN and you may get different result sets if you are not careful.</td>
</tr>
</table>

The two queries in the sample below return the exact same results and use the exact same execution plan.

Sample Subquery

```
SELECT CA.AccountID, CA.CustomerID, CA.AccountNumber
FROM CustomerAccount AS CA
    WHERE CustomerID IN (SELECT C.CustomerID
                         FROM Customer AS C
                         WHERE ZipCode LIKE '1%')
;
```

Sample JOIN

```
SELECT CA.AccountID, CA.CustomerID, CA.AccountNumber
FROM CustomerAccount AS CA
  INNER JOIN Customer AS C
  ON CA.CustomerID = C.CustomerID
WHERE C.ZipCode LIKE '1%'
;
```

The OVER clause you learned along with aggregate functions and the ranking function in Chapter 5 are other options to use instead of subqueries that are sometimes required due to the rules associated with the GROUP BY statement. Be careful though - the OVER clause can be very expensive in terms of performance.

In the following example, both queries produce the same result set. But when comparing execution plans, the query cost estimate places the subquery at a relative cost of 11% versus the OVER clause query taking 89% of the relative cost. This example is a reworking of the query used in the TRY IT 1 exercise earlier in this chapter.

Sample Subquery – much better performance

```
SELECT A.AccountID, A.OpeningDate, A.OpeningBalance
  , (SELECT COUNT(*)
       FROM LoanTransaction AS LT
       WHERE LT.AcctID = A.AccountID
       ) AS NumberTransactions
FROM Account AS A
WHERE AccountTypeID BETWEEN 6 AND 9
;
```

 Copyright © 2018 Ann L. Weber

Sample OVER Clause – significantly more resources used

```
SELECT DISTINCT A.AccountID, A.OpeningDate, A.OpeningBalance
   , COUNT(*) OVER (PARTITION BY A.AccountID)
           AS NumberTransactions
FROM Account AS A
   INNER JOIN LoanTransaction AS LT
   ON LT.AcctID = A.AccountID
WHERE AccountTypeID BETWEEN 6 AND 9
;
```

Real World!	Although you will find SQL practitioners and organizations that say you should always use one way of doing something or never use a particular command or tool, I have found that if performance is an issue, you should test multiple methods and use the one that consistently includes the fewest reads, the lowest estimated cost, and the lowest average CPU time when the command is run many times in equal situations. Derived tables, CTEs, temp tables, OVER clauses all have their pros and their cons. There are times when each will perform best.

Try It 4 – Reworking Subqueries

In the following exercise, you will rework an existing subquery into a join and test the results.

1. Open the **Try It 4 – Rework Subqueries Starter.sql** file from the **\Chapter 07 Subqueries\Try It Exercises** folder.
2. Click **File | Save As**, and then browse to the \Student Files folder. Type **Ch7Ti4.sql** in the File name, and then click **Save**.
3. Review and execute the query.
4. Rewrite the query as a join, leaving the initial query intact. The resulting query is shown below.

```
SELECT C.CustomerID, C.FirstName, C.LastName
     , CA.AccountNumber
FROM Customer AS C
     INNER JOIN CustomerAccount AS CA
     ON C.CustomerID = CA.CustomerID
WHERE C.ZipCode LIKE '1%'
;
```

5. Turn on the Include Actual Execution Plan (Ctrl + M) for the active query window.
6. Execute both queries at the same time. Review the results and the Graphical execution plans. Which execution plan is better?
7. Save and close your query file. Leave SSMS open for the lab.

Chapter 8 - Importing Data

In this chapter:

Import/Export Wizard
Exporting Data with the Wizard
Understanding Data Types
Common Import Concerns
Quality checking imported/exported data
Chapter 8 Lab
Answers to Exercises

Files needed:

- \Chapter 08 Importing\Inline Samples
- \Chapter 08 Importing \Try It Exercises
- \Chapter 08 Importing \Labs\
- \Student Files

	Some of the Try It exercises in this chapter build on one another, but are independent of other chapters. Answer files can be found in the \Chapter 08 Importing\Try It Exercises folder.

 Copyright © 2018 Ann L. Weber

Import/Export Wizard

The Import/Export Wizard provides you with the ability to easily import and export data to and from large of data structures. The Import/Export Wizard is a part of SQL Server Integration Services (SSIS) and can be used to create a starter package that you can modify and expand on by using SQL Server Data Tools (SSDT).

SSIS and SSDT are beyond the scope of this class. Numerous courses, books, and web sites are available on this topic. You can start at https://docs.microsoft.com/en-us/sql/integration-services/ssis-how-to-create-an-etl-package?view=sql-server-2017 and https://docs.microsoft.com/en-us/sql/integration-services/sql-server-integration-services?view=sql-server-2017.

You can launch the Import/Export Wizard from multiple locations, including the following:

- The Start menu, typically under the Microsoft SQL Server *version#* folder. It is titled SQL Server *version#* Import and Export Data and is available in both 32 and 64 bit versions.
- Inside of SSMS, in Object Explorer, right-click the database that will be your source or destination, then click **Tasks | Import Data** (if your database is the destination) or **Tasks | Export Data** (if your database is the source). Both options launch the same wizard. The only difference is the default database selected if you choose to use a SQL connection manager in the source (for the Export Data option) or the destination (for Import Data).
- Inside of SSDT, click Project | SSIS Import and Export Wizard.
- Inside of SSDT, in Solution Explorer, right-click the Packages folder, and click SSIS Import and Export Wizard.
- Run DTSWizard.exe from the command prompt.

The pages of the Import/Export Wizard along with an explanation of the options on each page are explained in the following sections. Each section includes a Try It exercise that covers the current page/ section. Each Try It builds on the one before. The only answer file available is the completed .dtsx package file.

Welcome Page

The welcome page, shown in Figure 92, is simply that and includes a check box that you can select to no longer see the welcome page when you relaunch the tool.

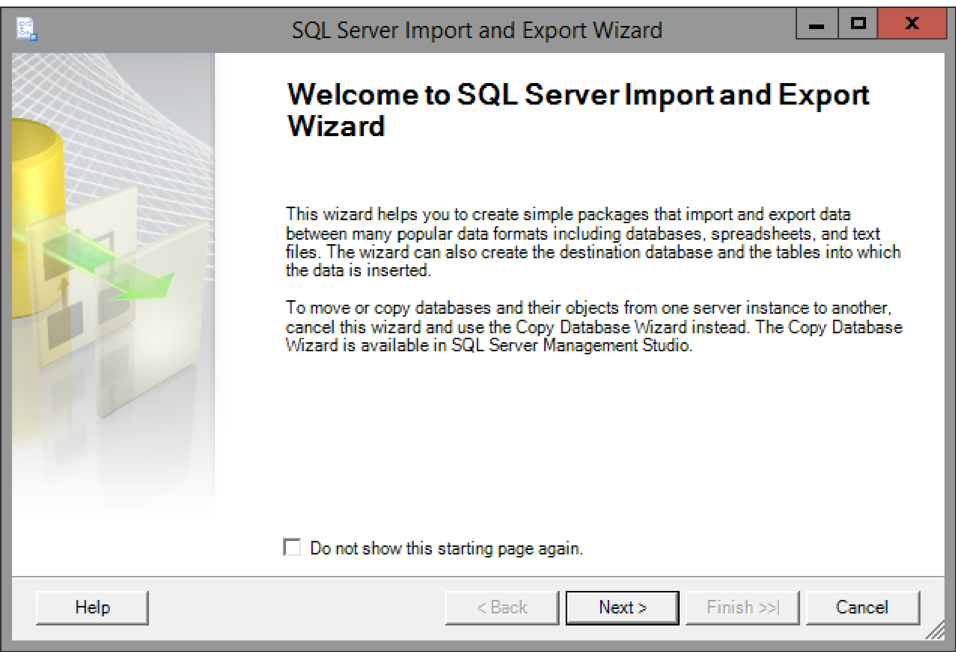


Figure 92: Welcome Screen

Choose a Data Source

All of the options on the Choose a Data Source page are dependent on the data source type (driver) selected in the drop-down list. The most common data sources include flat files (fixed width or delimited), Excel files, and other databases. A portion of the drop-down list is shown in Figure 93. For Microsoft SQL Servers version 2012 or later, you should choose the SQL Server Native Client 11.0 driver.

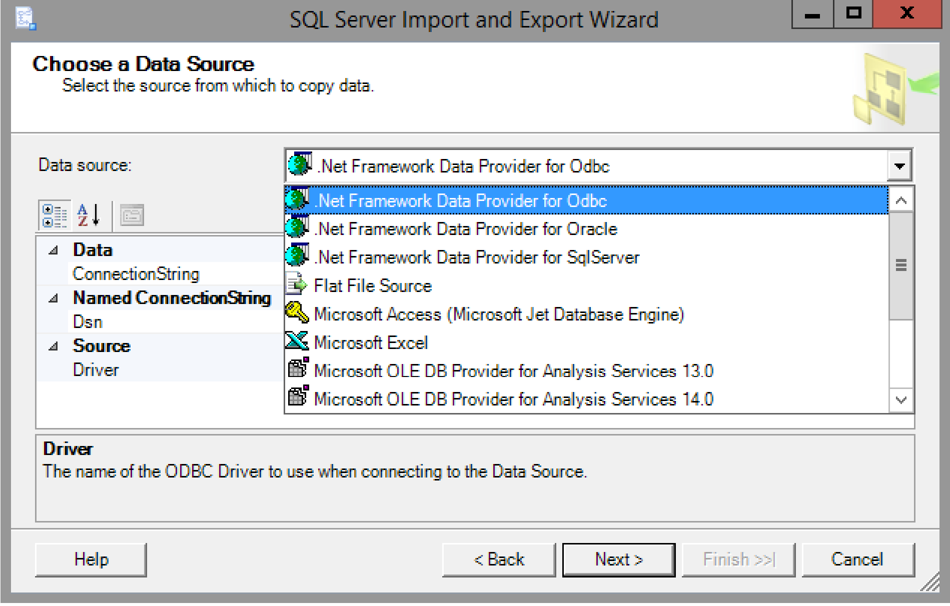


Figure 93: Data Source Dropdown

Once you select the data source driver type, the options of the page will change to reflect the required settings for that driver.

	Additional information on the support data sources and destinations along with links to additional help and tutorials can be found at https://docs.microsoft.com/en-us/sql/integration-services/import-export-data/import-and-export-data-with-the-sql-server-import-and-export-wizard?view=sql-server-2017.

Try It 1 – Starting the Import/Export Wizard

This exercise will be part of a series that will walk you step by step through the process of importing a csv file into the RetailBankingSample database. At the end of the series, you will have an SSIS package and additional rows in the Customer table.

	The final .dtsx package created through this series of Try It exercises can be found in the Chapter 08 Importing\Try It Exercises\Answers folder and will be named Try It 5.dtsx. It will include all of the steps from Try It exercises 1 to 5. You can open and execute this package in SSDT or Visual Studio if you have problems importing the data on your own.

1. Open SSMS.
2. In Object Explorer, right-click the RetailBankingSample database and click **Tasks | Import Data**.
3. On the Choose a Data Source page, select **Flat File Source**.
4. On the right side of the File Name box, click the Browse button.
5. Browse to **\Chapter 08 Importing\Try It Exercises\Starter**, select **CSV files (*.csv)** in the drop-down, and then click **NewCustomers.csv**, and then click **Open**.
6. Enter one double-quote (“) in the Text qualifier field, verify that Column names in the first data row is selected, and review the other options. Notice the warning at the bottom of the wizard explaining that you have not yet defined columns. The General page of the Choose a Data Source should look similar to Figure 94.

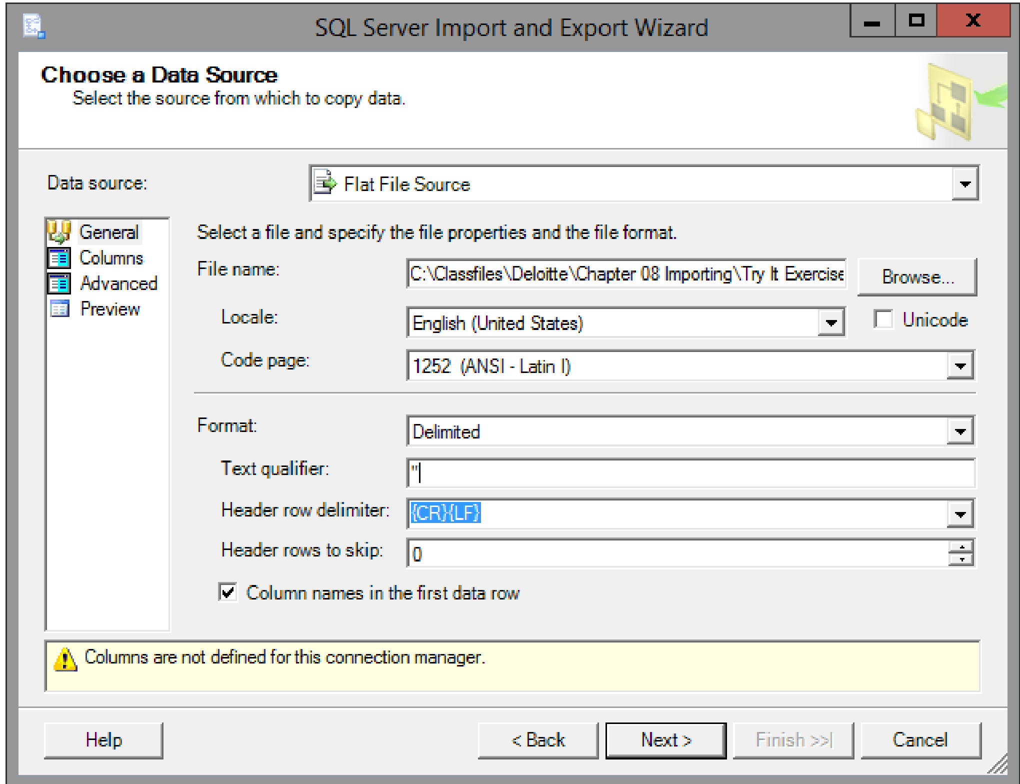


Figure 94: General Page

7. Click **Columns** in the page list on the left side of the Choose a Data Source page.
8. On the Columns page, review the column names and rows shown in the preview. If your column names have double-quotes (") around them, return to the general page, verify that you have one double-quote in the Text qualifier field, return to the Columns page, and then click **Reset Columns**.
9. Change to the **Advanced** page within the Choose a Data Source options.
10. On the Advanced page, with **CustomerID** highlighted, change the DataType to **four-byte signed integer (DT_I4)**.
    Note: If the data cannot be imported because of data entry problems with incorrect data types, you want the import to fail as soon as possible. For this reason, you will change the data types of the CustomerID and Birthdate columns to match the data types in the Customer table.
11. Click and select Birthdate and then change the DataType field to **database timestamp (DT_DBTIMESTAMP)**, and then click **Next**.
12. Leave SSMS and the Import/Export wizard open for the next Try It exercise.

 Copyright © 2018 Ann L. Weber

Choose a Destination

On the Choose a Destination page, you will be presented with list of available destination driver types. Like the Choose a Data Source page, the options on the page vary based on the driver type select.

When you select the SQL Native Client 11.0 driver, you will need to configure the SQL Server Instance, authentication information and destination database. Optionally, you can create a new database as the destination.

To create a new database as part of the Import/Export Wizard process, click the New button next to the Database selection box on the Choose a Destination page. Figure 95 shows the Create Database dialog box. You can configure the database name along with data and log file size and auto-growth settings. You cannot add multiple files, file groups, or customize the file names. If you need to configure any of these optional features you should create the new database before starting the Import/Export Wizard.

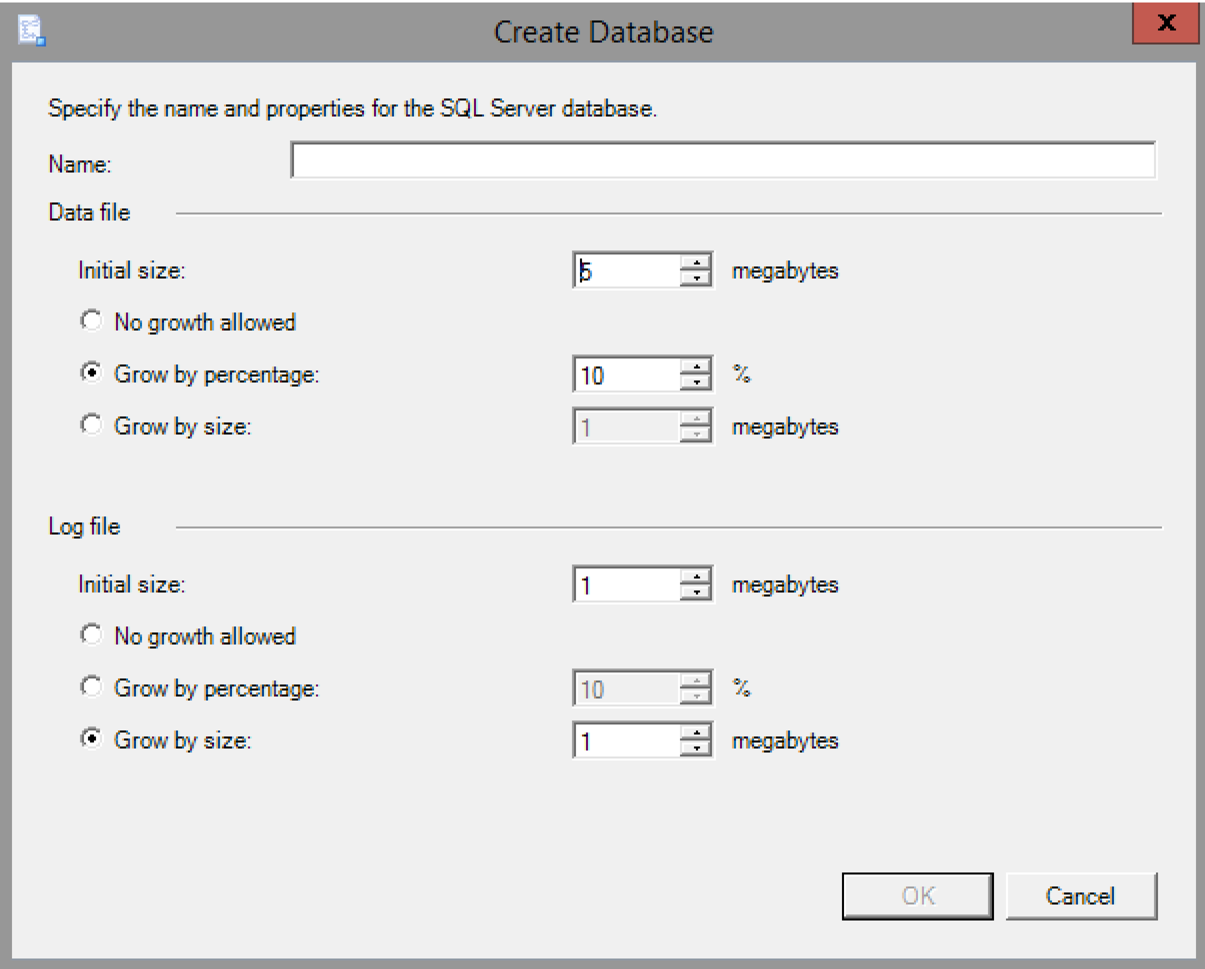


Figure 95: Create Database Window

When you select a flat file destination, you will need to define the file type, location, delimiters, and more, similar to the flat file source options you configured in Try It 1.

Try It 2 – Choose a Destination

In this exercise, you will continuing configuring the package to import new customer information by configuring the Choose a Destination page with the Customer table in the RetailBankingSample database as the destination.

1. On the Choose a Destination page, select SQL Server Native Client 11.0 in the Destination drop-down list.
2. Verify that (local) or your SQL Server's instance name is typed in the Server name box. To avoid potentially long waits, avoid using the drop-down box for this option.
3. Use the Drop-down box to select the **RetailBankingSample** database, and then click **Next**.
   Note: When you launch the wizard from outside of SSMS, using the drop-down list will verify the connectivity with your SQL Server instance.
4. Leave SSMS and the Import/Export wizard open for the next Try It exercise.

Specify Table Copy or Query

When your source is a database rather than a flat file, you will have the option to select from a list of tables and views in the source or to write a query to define what data will be imported.

Best Practice!	Write a select statement that only retrieves the columns and rows that you need. Even though you can choose to "ignore" columns later in the process, these extra columns slow down the retrieval and add more pressure on the memory.

Destination Configuration page

Depending on your destination, this page will have a different name and options.

For a SQL Server destination, the page shown in Figure 96 looks very straight forward with only a Source column and a Destination column specifying the table, query, or file used as the source and the destination and is called the Select Source Tables and Views page.

 Copyright © 2018 Ann L. Weber

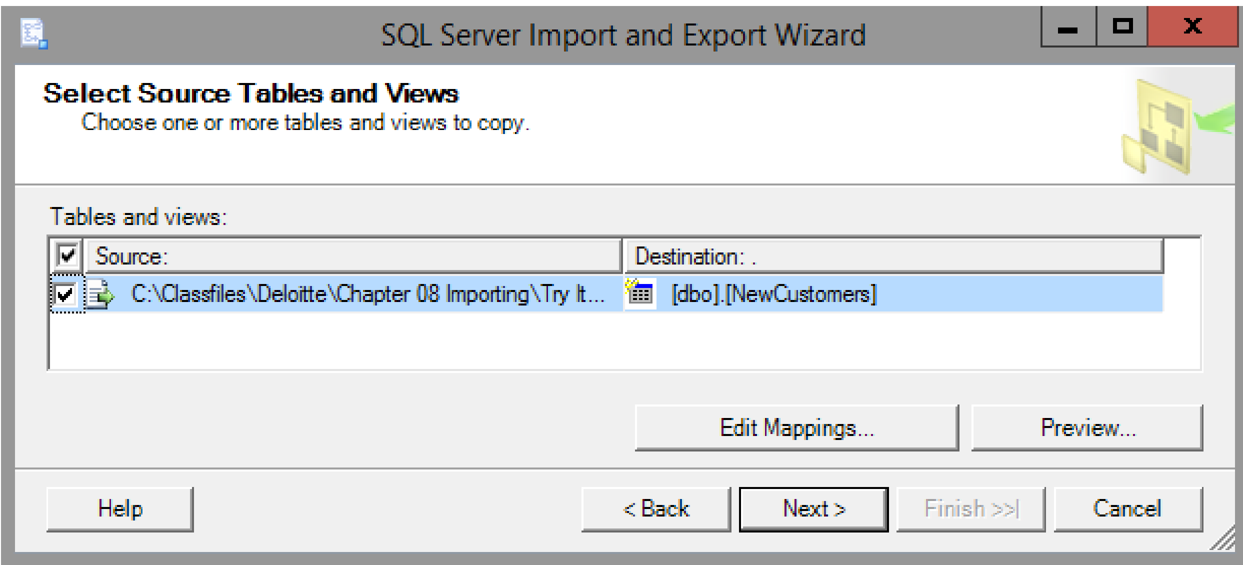


Figure 96: Select Source Tables and Views

If you want to select a different destination table, click in the box to activate it. You can then either select a table from the drop-down menu, or place your cursor inside of the box and type a new name.

For a flat file destination, the Configure Flat File Destination shown in Figure 97 allows you to select the row and column delimiters for the file. Additionally, if you are copying from a table rather than writing a query, there is a drop-down list to select the source table.

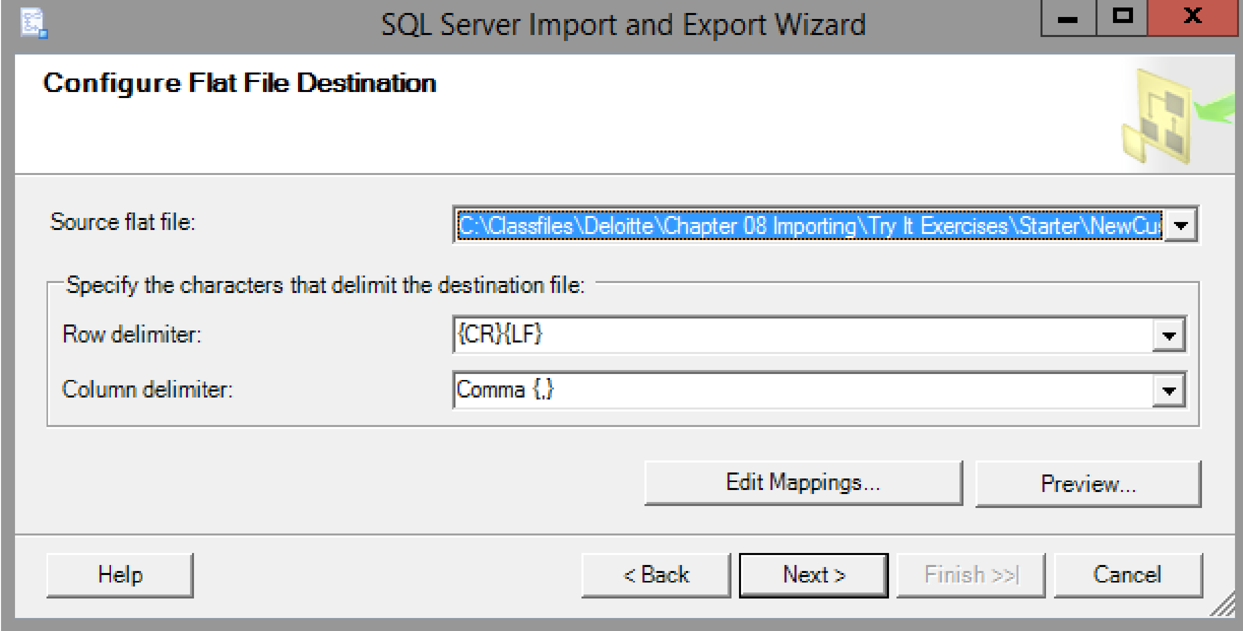


Figure 97: Configure Flat File Destination

The Edit Mappings and Preview buttons are consistent across data destinations, but the Edit Mappings options presented may be different. The Preview button allows you to look at the data as it is currently

configured. If one of the columns has been defined incorrectly, you may be able to catch it here rather than later in the process.

When you click the Edit Mappings button you are presented with a number of options depending on the destination type.

With a SQL Server destination where the destination table already exists, you have the option to Delete rows in destination table (effectively replacing the existing rows with the new rows) or Append rows to the destination table (adding the rows to those already there.) By default, the Append rows option is selected.

Alternately, if you were creating a new table in the database with the wizard, the Create destination table would be selected, the append and delete options would be grayed out since there is no existing data, and the Drop and re-create destination table option could be selected. The drop and re-create option allows you to import the full set of data every time without worrying about what portion of the data is new and needs to be inserted.

The Enable identity insert option allows you to manually define values for a column that normally auto increments with each new row. Be careful with enabling this option because you may end up with duplicate keys of other data problems.

The mappings section is the same regardless of the destination type and shows what source columns will be mapped to what destination columns and what the destination data type will be. If the column names are not spelled exactly the same, the columns will not be mapped correctly and you will need to use the drop-down lists that appear when you click in a column as shown in Figure 98.

 Copyright © 2018 Ann L. Weber

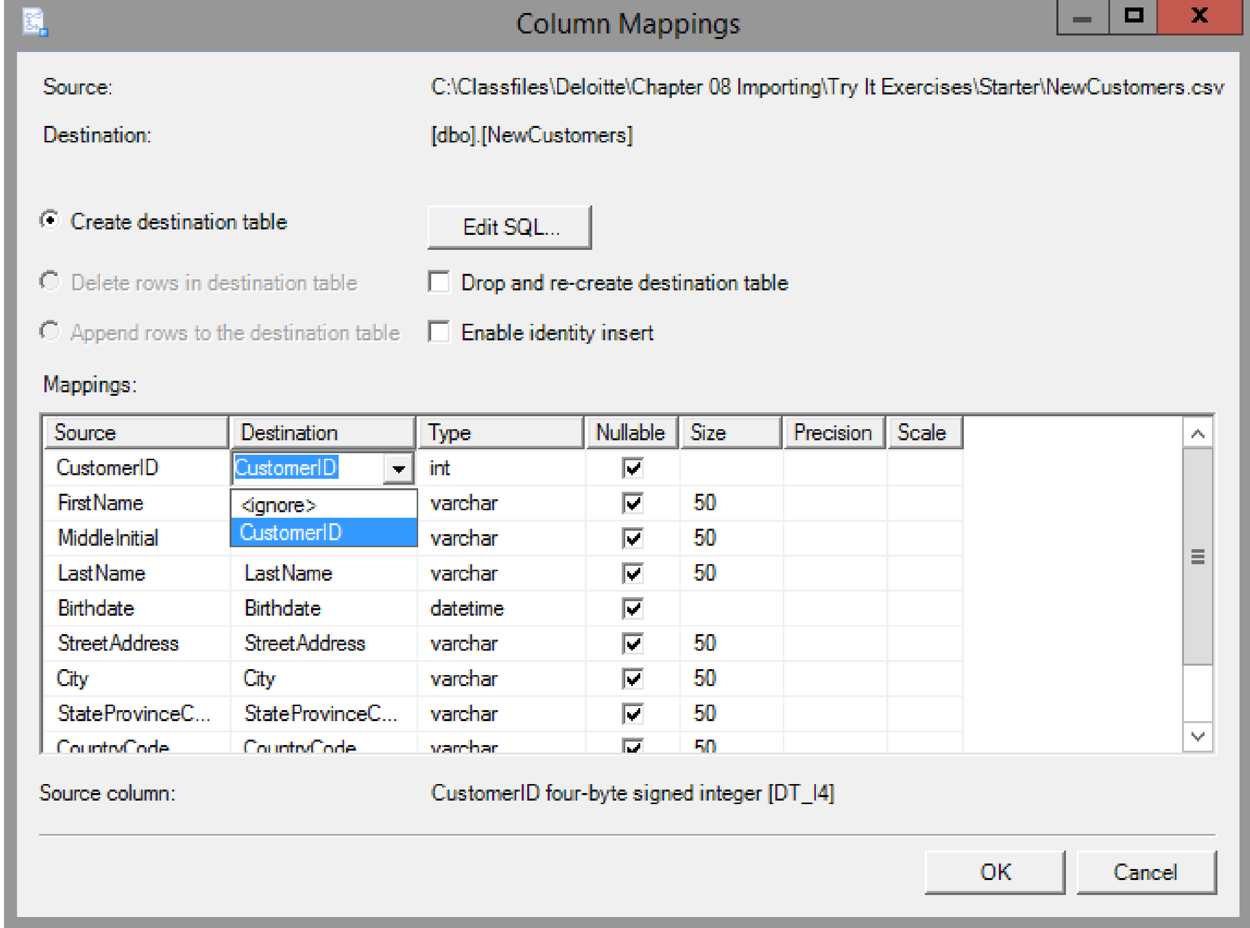


Figure 98: Column Mappings

Also, as shown in Figure 98, you can choose the <ignore> option to leave the destination field blank. On a SQL or other database destination, if the destination does not either allow NULL values or have a DEFAULT constraint defined for the column, selecting <ignore> will cause the import to fail.

Like the database destination, the Column Mappings for the flat file destination defined as a new file, the only option available will be the Create a destination file option and it will already be selected for you. If the file already exists, you have the option to delete and replace the existing rows or append to them.

	Each Destination column can only be mapped once. When you select a destination column for one source column, that destination column is removed from the drop-down list. If everything has been mapped, but you need to reverse two mappings, you must set one of the mappings to <ignore> and then change the other mapping. Once the initial required destination column is free, you can change the <ignore> to the valid column.

Try It 3 – Select Source Tables and Views

In this exercise we continue creating our package by defining the existing Customer table as our destination and verifying that the new rows will be appended to the existing rows rather than replacing them.

1. On the Select Source Tables and Views, select **[dbo].[Customer]** from the Destination drop-down list.
2. Click **Edit Mappings**.
3. Verify that **Append rows to the destination table** is selected. The other option will delete the original data before adding the new rows.
4. Review the data types being inserted and then click **OK**.
5. Click **Next**.
6. Leave SSMS and the Import/Export wizard open for the next Try It exercise.

Review Data Type Mapping

For database destinations, the Review Data Type Mappings page will show you a summary of your source and destination choices, including each column's source and destination name and data type. It also will include any information on required conversions along with any warnings or errors.

The final configuration options on this page define how you want the process to react when either errors or data truncation occur. By default, both types of "errors" will cause the entire package (import or export) to fail. You can set both the On Error and On Truncation handling independently for each column, or you can define the behavior for all columns by using the global settings near the bottom of the page as shown in Figure 99.

 Copyright © 2018 Ann L. Weber

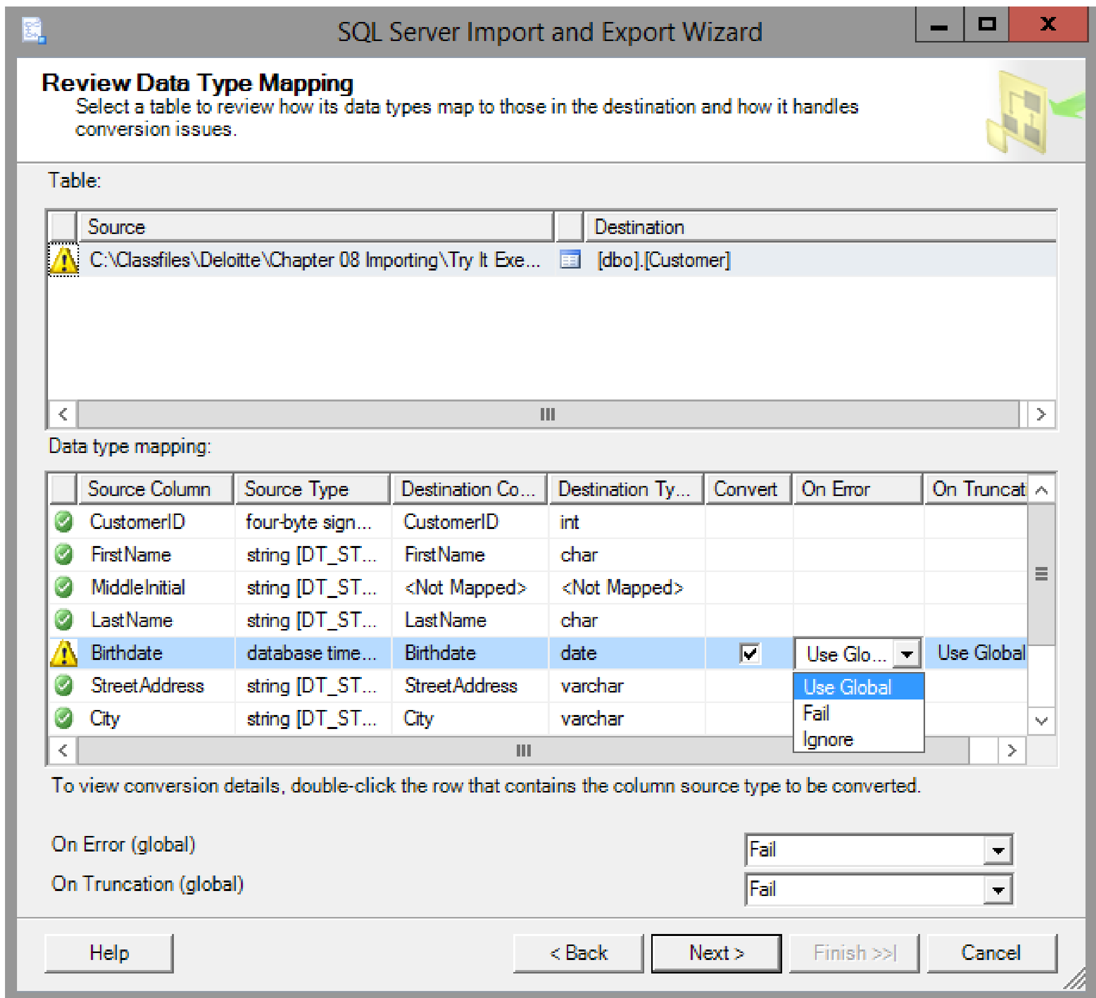


Figure 99: Review Data Type Mapping

Truncation can occur with almost any data type when the data to be inserted is outside of the scope of the defined data type. For example, when a value made up of 50 characters is sent to a destination column that only supports 20 or a numeric field with 8 digits to the right of a decimal point is placed into a column with a currency data type which only support 4 digits to the right of the decimal point. Truncation can occur at any step within the package.

Errors can come from numerous problems including duplicate primary key fields, a hire date of Feb 31, 2018 being converted to a date field, a letter in a field where the destination data type in integer, a NULL value being placed in a destination column that does not allow NULLs, and much more.

Other destinations such as a Flat File destination will take you straight to the Save and Run Package page.

Try It 4 – Review Data Type Mapping

In this exercise you will review the options provided and the current configuration without making any changes.

1. On the Review Data Type Mapping page, notice the warnings in the Table section at the top and next to Birthdate in the bottom section. This is because we defined the date information to be imported as a data type that includes date and time, but the database is using the date data type. Since the dates do not have time associated with them, truncation will not occur and the file should import successfully.
2. Review the options available when you click the drop-down list on Birthdate row under the On Error column. Review the drop-down list options for the global settings at the bottom of the page.
3. Click **Next**.
4. Leave the wizard and SSMS open for the next Try It exercise.

Save and Run Package

If you are familiar with editing packages in SSDT or running a package independently from the command prompt with DTEXEC.exe, you can choose to save the SSIS Package. When you click the Save SSIS Package button, you have the option to save the package to the msdb database on the SQL Server or to the file system as a .dtsx file.

Once you decide to save the file, you will need to define how any usernames and passwords that were included in the connection strings will be secured if they are to be stored as part of the package. Additionally, if you choose to save your package, the Save SSIS Package page will appear next rather

 Copyright © 2018 Ann L. Weber

than the Complete the Wizard page.

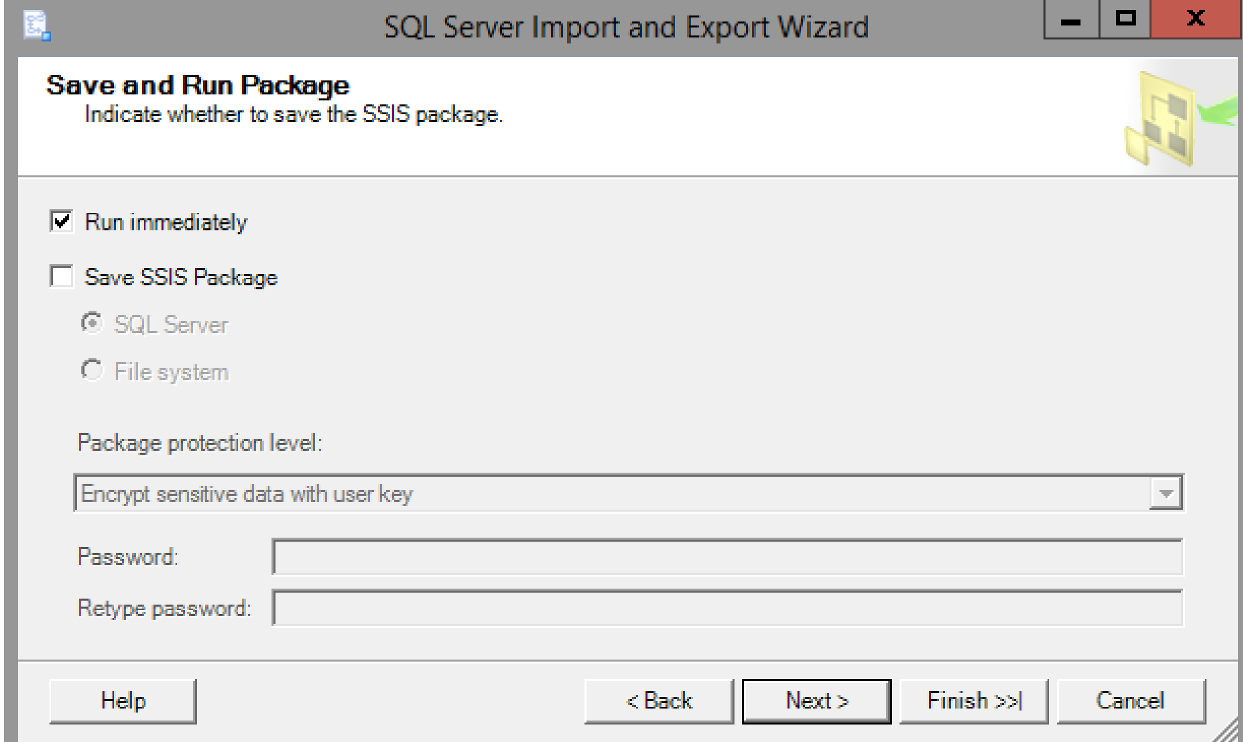


Figure 100: Save and Run Package

Saving packages and the security settings involved are beyond the scope of this class. You can read more about these features at https://docs.microsoft.com/en-us/sql/integration-services/import-export-data/save-and-run-package-sql-server-import-and-export-wizard?view=sql-server-2017.

Complete the Wizard

The Complete the Wizard page allows you to review the steps that the package will perform. It also provides you with a Back button so that you can go back to the previous steps and fix any problems that you notice during your review.

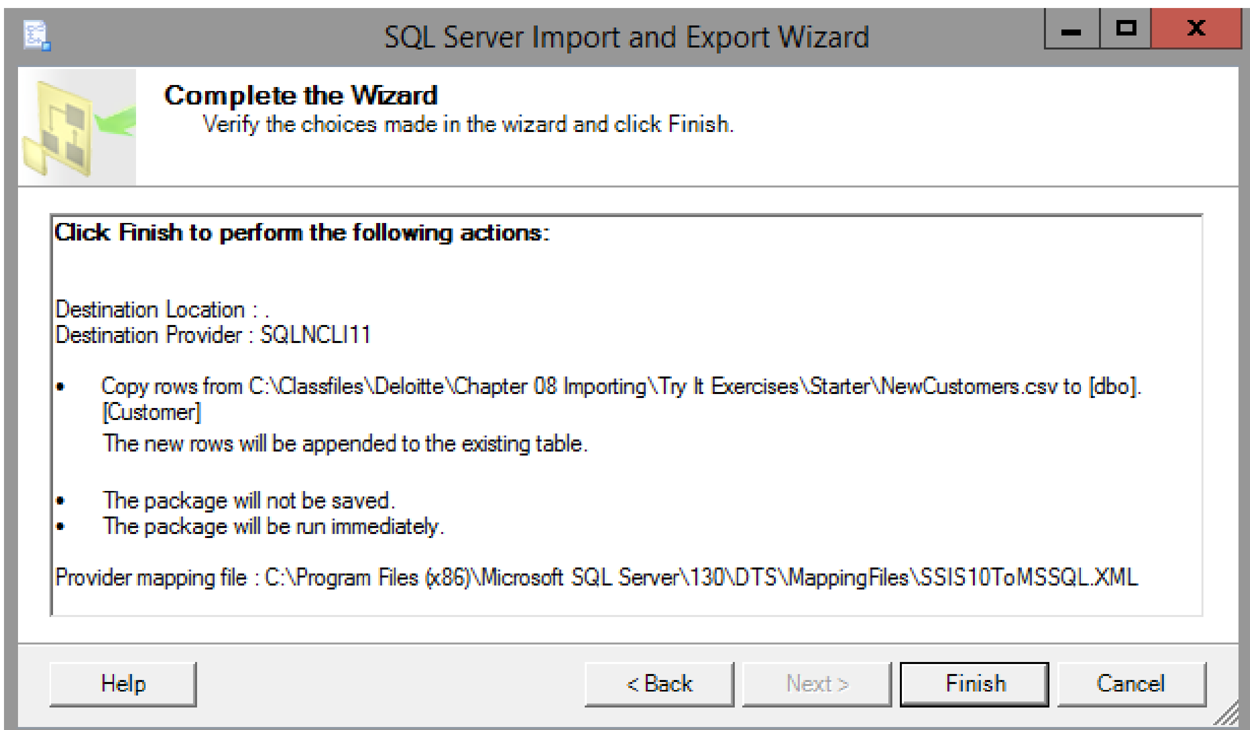


Figure 101: Complete the Wizard

Execution Results Page

The title at the top of the page changes depending on the state of the package, running, successful, or failed. This page shows you step by step what occurred during the execution, including row counts. You should review warnings, but typically, especially in the validation stage, a warning without an associated error is not a problem.

 Copyright © 2018 Ann L. Weber

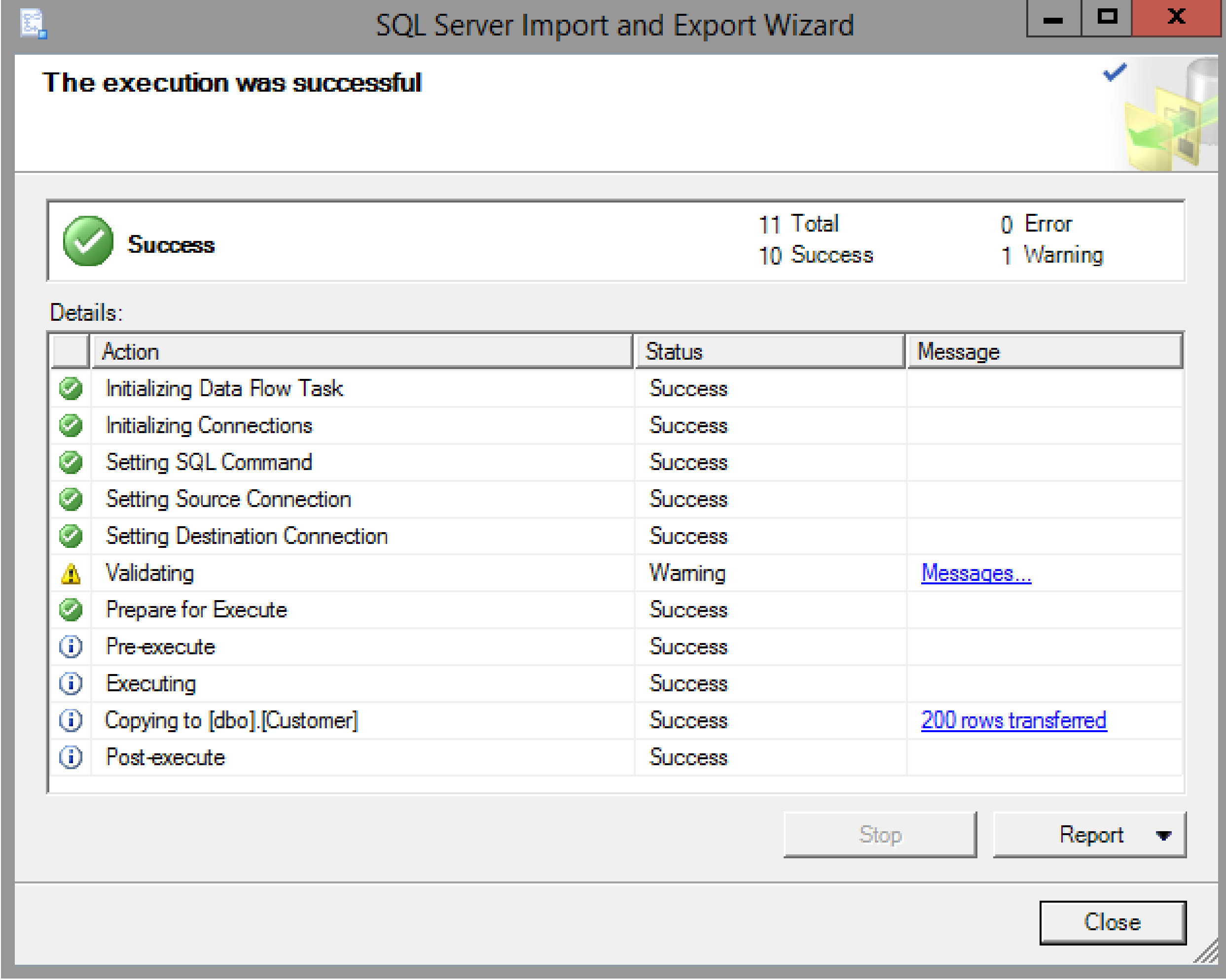


Figure 102: Execution Results

The links in the Message column can be used to provide additional information. Also, if you would like to see more details or have the results available after you click Close, you have four options by clicking the Report button including:

- View Report
- Save Report to File
- Copy Reports to Clipboard
- Send Reports as E-mail

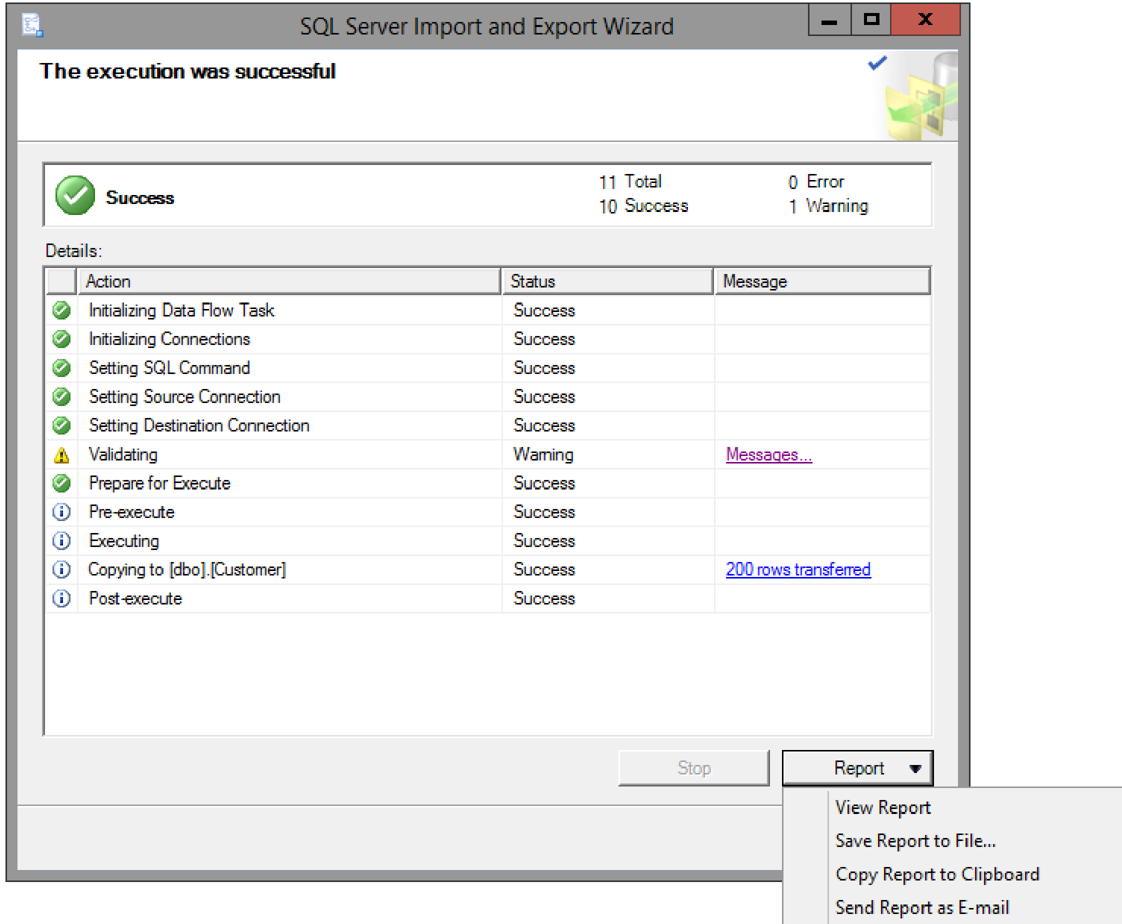


Figure 103: Report Options

If anything fails during the import, the Back button will still be active to go back and try again. Once the package succeeds, the Back button is disabled, so unless you saved the package, you cannot run it again without redefining it.

Try It 5 – Completing the Wizard

In this exercise you will complete and run the import package that you have been defining throughout this series.

1. On the Save and Run Package page, verify that only **Run immediately** is selected and then click **Next**.
2. On the Complete the Wizard page, review the steps that will be performed, notice that you can click the Back button to fix anything that wasn't configured correctly, and then click **Finish**.
3. On the results page, verify that 200 rows were transferred. Review any other messages.

 Copyright © 2018 Ann L. Weber

4. Click **Close**.

Exporting Data with the Wizard

Exporting data is really not any different from importing. The difference is that typically when we talk about exporting, we are starting with a SQL Server database and exporting to flat files, Excel files, or other destinations. However, the wizard does not require a SQL server database to be used and can be used to move data from a flat file to an Excel file to while performing some data conversions along the way. In this case, there is no difference at all between an import and an export. You can select either option and the one you select is completely irrelevant.

Try It 6 – Exporting Data from SQL to Excel

In this exercise, you will take the new combined results from the Customer table and export them to a new Excel file named AllCustomers.xlsx and saved in the \Student Files folder.
Note: If you did not perform Try It exercises 1 through 5, you can double-click and run the \Chapter 08 Importing\Labs\Starter\RunPackage.bat file.

1. In SSMS, in Object Explorer, right-click the RetailBankingSample database, and then click **Tasks | Export Data**.
2. If the **Welcome** page appears, click **Next**.
3. On the Choose a Data Source page, select **SQL Server Native Client 11.0** from the drop-down list.
4. On the Choose a Data Source page, verify that your SQL Server name, Authentication, and the RetailBankingSample database options were all populated correctly and then click **Next**.
5. On the Choose a Destination page, select Microsoft Excel.
6. Click **Browse** next to the Excel file path and browse to the **\Student Files** folder. Type **AllCustomers.xlsx** in the file name, and then click **Open**.
7. The Excel version should automatically update to Microsoft Excel 2007. Verify that First row has column names is selected and then click **Next**.
8. On the Specify Table Copy or Query page, verify that Copy data from one or more tables or views is selected, and then click **Next**.
9. On the SELECT Source Tables and Views, click to select the dbo.Customer table.
10. Change the table (tab) name in the destination column to `AllCustomers`. If you accidentally erase the accent marks, it is the one under the tilde to the left of the number 1 on the keyboard.
11. Click Edit Mappings. Notice that the options are similar to the database options. In Excel, creating a table equates to creating new tab in the workbook. Dropping a table deletes the tab in Excel.
12. Change the **MiddleName** data type to **VarChar** with a size of **20**. Verify that ZipCode is VarChar, and then click **OK**.

13. Click Preview and review the information.
14. Click OK, and then click **Next**.
15. On the Review Data Type Mapping page, double-click the yellow triangle next to FirstName and review the conversion warnings. Leave the values are they are and click **Next**.
16. Click Next on the Save and Run Package page.
17. Review the information on the Complete the Wizard page and then click Finish.
18. Verify that 500 rows were successfully transferred, and then click **Close**.
19. Leave SSMS open for the Lab.

Understanding Data Types

Selecting the correct data types when defining the SQL tables to receive your imports as well as defining data types for imported files and any required data conversions is very important. If you try and put a zip code into an integer field, you will lose all of the leading zeros. If you have three billion records and you try to import them into the integer (int in T-SQL) or four byte signed integer (DT_I4 in SSIS) column, an error will occur.

Not only should you carefully consider the data types that you (or the wizard) are picking for different columns, you should also check your data carefully after an import to verify that nothing was lost or truncated unexpectedly.

	You already learned about the majority of the common data types for SQL Server in Chapter 3 Built-in Functions Overview. For more information on each SSIS data type and the values that it can accept see https://docs.microsoft.com/en-us/sql/integration-services/data-flow/integration-services-data-types?view=sql-server-2017#mapping-of-integration-services-data-types-to-database-data-types. Also, within that very large help page, you can jump directly to https://docs.microsoft.com/en-us/sql/integration-services/data-flow/integration-services-data-types?view=sql-server-2017#mapping-of-integration-services-data-types-to-database-data-types to see how the SSIS data types map to SQL Server and other data types.

Common Import Concerns

There are number of things that can either go wrong, or that you need to be aware of during imports. One of these concerns is working with different data types and how SSIS determines what data types to use. Additionally, every type of source or destination has its own data types that may or may not convert nicely to other system's data types.

 Copyright © 2018 Ann L. Weber

Truncations and Data Type Conversions (Implicit and Explicit)

In our Try It exercises at the beginning of the chapter, we imported new first and last names into our table. SSIS picks a size of 50 characters when importing from a flat file as the size for the data string in SSIS. The column in the database only supports up to 20 characters. If you set SSIS to ignore truncation messages, you can end up with partial names without knowing about. You have quite a few options here including:

- Decide that you don't care about truncations
- Set the package to fail on truncation
- Ignore truncations during the package, retrieve all of the rows with a character length of 20 and review those fields against those on the original source file.
- Import everything into a new table where every field is set to an extremely large character size and test the data before moving it into the final destination table with the correct data types.

	In my experience, the data type conversions that seem to cause the greatest difficulty come from importing and exporting with Excel files. Although the Wizard typically does a pretty good job handling these conversions, there will be times when you need to step in and manually make changes beyond how the wizard configures the conversions.

Another concern with data types is selecting the wrong data type for the destination table when you create the table during the import. In addition to the leading zero disappearing in integer fields that has already mentioned, and data truncation, you may also run into problems using date functions if you import your dates as strings. You won't be able to use string functions if you import your strings to the text data type instead of one of the character data types. You will have trailing spaces if you use fixed character vs variable length character fields and the list goes on. Luckily, in development, (where all of these things should be done and thoroughly tested first) if you still have the original csv file and you imported something only to find that it doesn't meet your needs, you can drop the table and import it again. That is why you want to do thorough testing before moving to the next step in the process.

Date fields

There are a number of things to be careful with when you working with dates. First of all, there are circumstances where empty date fields are converted to Dec 30, 1899 or January 1, 1753. When you import dates, you should verify that you do not end up with incorrect Dec 30, 1899 and January 1, 1753 values in your data. There are work arounds in the full SSIS if you are running into these situations. If you are not familiar with SSIS, you can remove all empty values from string fields before importing. For example, some companies pick Dec 31, 9999 as a date to enter when a valid date is not available.

NULLS

Like in other parts of SQL, NULL values require some special understanding and handling. If you are trying to import rows of data that include missing values in columns that do not allow NULL values, you may need to use the full SSIS, import into a table that allows NULLs first and then move them with a query using the ISNULL function to replace the NULL values, or some other work around. With some data types, the wizard automatically picks a value for you and converts the empty field to this value.

Finding a Good Delimiter

The comma is the most common field delimiter, but commas are frequently found in many companies data as well. When you don't have a good field delimiter, you must use double quotes or some other text qualifier around each field, making the text files harder to read.

SSIS and the Import/Export Wizard support the vertical bar (| known also as Pipe), which typically is a safe delimiter since it rarely appears in data.

	If you choose to use the vertical bar when exporting, Excel cannot automatically interpret the delimiter if you double-click the xlsx file to open it. You can, however, successfully separate the columns in a couple of different ways. You can open the file, highlight the contents, and then on the Data tab, select Text to Columns, select Delimited on the first tab, and then click Next. On the following page, change the Delimiters from Tab to Other and put the Pipe {|} symbol in the empty box. Validate the data preview and then click Finish. You can also create a new blank workbook in Excel and use the From Text option on the Data tab, in the Get External Data section.

Quality checking imported/exported data

Once your import is complete, you will want to perform some sort of quality checking. Depending on your environment, company policies, what is happening next, what will the data be used for, and more, the complexity and thoroughness of these checks can vary greatly.

When running these checks, you will need to determine if any problems you find are coming from the underlying data or from the import process.

In general, you should perform the following tasks as part of your import validation:

- Compare imported totals to report totals if available
- Validate any date columns, looking for Dec 30, 1899 and January 1, 1753 values, blank fields, or out of range data.
- Verify that any NULL values were actually empty strings and not lost data
- Check string field lengths to try and see if data was truncated during the import, especially if you turn off the Fail on truncation options in the Wizard.
- Verify that the right data ended up in the correct columns. This is especially important when the data has commas and a comma was chosen for the delimiter.

 Copyright © 2018 Ann L. Weber

- Verify leading 0s weren't lost from fields that should be stored as characters and not numbers such as account numbers and zip codes.

Try It 7 – Validating Data

In this exercise, you will review the data that you imported in the series of Try It exercises numbers 1 through 5 and then exported in Try It 6. Although the process of verifying data will differ in each situation, this exercise walks you through one possible set of steps for verifying and troubleshooting. In this particular case, the problem is very obvious, making the steps we take seem like overkill, but you will gain exposure to all of the steps of the process.

1. Use Excel to open the \Student Files\AllCustomers.xlsx file that you created in Try It 6. If you did not perform the Try It exercise, you can locate the file in the \Chapter 08 Importing\Try It Exercises\Answers folder.
2. Use the Sort option on the Data tab to sort the Birthdate column. If necessary, click the box next to "My data has headers" to see the names of the columns.
3. If the Sort Warning appears, click Sort anything that looks like a number, as a number, and then click OK.
4. Review and document the following results.
    a. What is the most recent birthdate?
    b. What is the earliest birthdate?
    c. How many rows have empty birthdates?
    d. How many rows have a value of January 1, 1753?
    e. How many of the rows with a value of January 1, 1753 are from the new import?
    f. How many NULL birthdate values are for the newly imported rows?
5. Open a query window and write several SQL Statements to determine the answers to the questions listed above for the database. Are there any variances? This step validates that for the birthdate field, the Excel file and SQL have the same data.
   If you need help with the queries, check the **Try It 7 – Validating Data Answers.sql** file.
6. Use Notepad, Notepad ++, or another similar tool to open the \Chapter 08 Importing\Try It Exercises\Starter\NewCustomers.csv file with a program that will NOT automatically format and break apart the data.
7. Use Ctrl + F or another similar find option to look for the year 1753 in the data. Notice that there are not any rows in the source data with a date of 1753.
8. Compare the Excel file to the CSV file. Use the Excel file to locate the CustomerID of one or two of the rows with a birthdate of January 1, 1753. Locate that CustomerID in the CSV file. Notice that the date field was empty (just two commas side by side) in the CSV file.

9. Type and execute the following command to correctly set the NULL values for the dates that the Wizard converted to January 1, 1753. The Messages tab should state **"(10 row(s) affected)."**

   ```
   UPDATE Customer
   SET Birthdate = NULL
   WHERE Birthdate = '17530101'
   ;
   ```

   Note: You will learn about the UPDATE statement in **Chapter 9 Data Manipulation Language**.
10. Return to the Excel file and sort the information on the ZipCode column. Select the Sort anything that looks like a number, as a number option, and then click OK. At the top of the list, notice that some of the ZipCode fields are only 4 digits long while most are 5 digits long. Notice that all of the 4 digit zip codes have a CustomerID of greater than 300, making them part of the newly imported rows.
11. Compare the Excel file to the file open in Notepad or similar program. Use the CustomerID field to see the ZipCode value in the original CSV. Notice that the leading 0's for zipcodes in locations like MA, JH, etc that start with 0 are all missing. The program that generated the csv treated this field like a number and lost the leading 0s.
12. You need to fix the 0s in the database version of the data. Write a SELECT statement that will return every row from the Customer table where the length of the ZipCode field is 4 characters.
13. Type and execute the following UPDATE statement to fix the zip code values where the leading 0 had been dropped. 13 rows should be affected.

   ```
   UPDATE Customer
   SET ZipCode = '0' + ZipCode
   WHERE LEN(ZipCode) = 4
   ;
   ```

14. Save and close the query tab and both the Excel and CSV files. Leave SSMS open.

 Copyright © 2018 Ann L. Weber

Chapter 9 - Data Manipulation Language

In this chapter:

Transaction Overview
Insert
INSERT SELECT vs SELECT INTO
Update
DELETE
Chapter 9 Lab
Answers to Exercises

Files needed:

- \Chapter 09 DML\Inline Samples
- \Chapter 09 DML \Try It Exercises
- \Chapter 09 DML \Labs\
- \Student Files

	Some of the Try It exercises in this chapter build on one another. They independent of other chapters. Completed Try It queries can be found in the \Chapter 09 DML\Try It Exercises folder.

Data manipulation language (DML) is used to make changes to the data in your database. In this chapter you will learn the three data manipulation commands: INSERT, UPDATE, and DELETE. Now that you have learned to write SELECT statements and use functions, you can apply what you have learned to the DML statements.

Transaction Overview

SQL Server uses built-in locking to protect against "dirty" data. One example is the protection against viewing a modified record in cache that later gets rolled back. This is referred to as a dirty read. Although locks protect against write conflicts between multiple commands, locks do not control when a change is committed to the database. SQL Server uses transactions to manage when changes are written to the database and become permanent.

A transaction is a set of commands that succeed or fail as a unit. By default, SQL Server handles every statement that modifies data, schema or security independently as its own transaction. Because of this default behavior, if you accidentally delete all of the rows in your table, there is no undo or rollback behavior. Every DML, DDL, or DCL command that makes changes is automatically committed as soon as it finishes running.

	A full discussion of locks and transactions is beyond the scope of this class. For more information on transactions, see https://docs.microsoft.com/en-us/sql/t-sql/language-elements/transactions-transact-sql?view=sql-server-2017 and https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-2017.

Using Transactions

Transact-SQL uses four statements to manage transactions:

- BEGIN TRANSACTION
- COMMIT TRANSACTION
- SAVE TRANSACTION
- ROLLBACK TRANSACTION

BEGIN TRANSACTION

The BEGIN TRANSACTION statement starts a new transaction.

Syntax

```
BEGIN {TRAN|TRANSACTION}
    [{transaction_name|@tran_name_variable}
      [WITH MARK ['description']]
    ]
```

As the name implies, the BEGIN TRANSACTION statement starts a new transaction. From this point on, all statements are part of the transaction until you execute either a COMMIT TRANSACTION statement

 Copyright © 2018 Ann L. Weber

(which ends the transaction and makes the changes a permanent part of the database) or a ROLLBACK TRANSACTION statement (which ends the transaction and throws the changes away).

The WITH MARK clause ties transactions to the SQL Server transaction log, which saves a record of all database transactions. If you mark a transaction, you can later restore the log to the named mark. This can be an important consideration in developing a backup and restore strategy for your server.

COMMIT TRANSACTION

The COMMIT TRANSACTION statement is straightforward.

Syntax

```
COMMIT [{TRAN|TRANSACTION}
  [transaction_name|@tran_name_variable] ]
```

You'll notice that the transaction statements let you supply a name for your transactions, either directly or in a variable. These names can be up to 32 characters long and are intended as a readability aid to let you match up corresponding transaction statements in code. But there's an important caveat here: SQL Server doesn't actually pay attention to these names. A COMMIT TRANSACTION statement always refers to the most recent BEGIN TRANSACTION statement, regardless of any name you might assign.

SQL Server allows nesting transactions, but it does not behave as many expect. The only COMMIT statement that really matters is the final COMMIT because a ROLLBACK command rolls back all statements back to the beginning of the outermost BEGIN TRANSACTION statement. Because of this behavior, nesting of transactions is not recommended and should be avoided when possible.

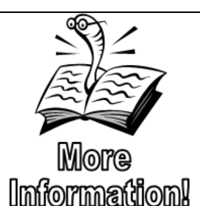
SAVE TRANSACTION

The SAVE TRANSACTION statement sets a *savepoint* within a transaction.

Syntax

```
SAVE {TRAN|TRANSACTION}
  {savepoint_name|@savepoint_variable}
```

Savepoint names can be up to 32 characters long. Setting a savepoint lets you create an intermediate spot within the transaction where the transaction can be partially rolled-back.



Savepoints and nested transactions are beyond the scope of this course. The **Extra Samples 09.sql** file in the **\Chapter 09 DML \Inline Samples** folder includes documented samples demonstrating savepoints and nested transaction behavior.

ROLLBACK TRANSACTION

The ROLLBACK TRANSACTION statement undoes some or all of the work in a transaction.

Syntax

```
ROLLBACK [{TRAN|TRANSACTION}
  [transaction_name|@tran_name_variable
  |savepoint_name|@savepoint_variable] ]
```

If you specify a savepoint name, the transaction is rolled back only as far as that savepoint. Otherwise, the transaction is rolled back to the beginning.

Sample

```
BEGIN TRAN;

UPDATE Customer SET LastName = 'Smith';

ROLLBACK;
```

Try It 1 – Using Transactions

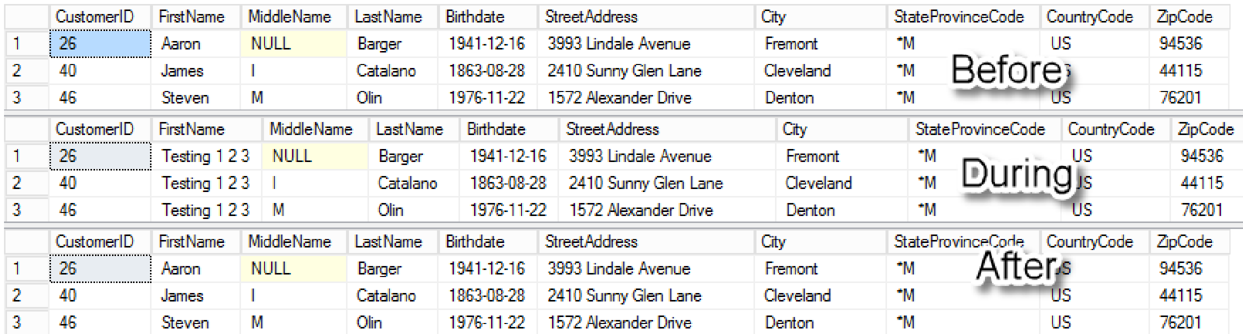
In this exercise you will update the data in the Customer table from within a transaction. You will then rollback the transaction comparing the state of the data from within the transaction and then again after it is rolled back.

Add a new Query Editor tab and save your new query as **\Student Files\SampleTransaction.sql**.
Set the database context to RetailBankingSample.
Write and execute the following query:

```
SELECT * FROM Customer
    WHERE StateProvinceCode = '*M';
BEGIN TRANSACTION
  UPDATE Customer
    SET FirstName = 'Testing 1 2 3'
    WHERE StateProvinceCode = '*M';
  SELECT * FROM Customer
    WHERE StateProvinceCode = '*M';
ROLLBACK TRANSACTION
  SELECT * FROM Customer
    WHERE StateProvinceCode = '*M';
```

Review the differences in the first and second result sets shown in Figure 104.

 Copyright © 2018 Ann L. Weber



	CustomerID	FirstName	MiddleName	LastName	Birthdate	StreetAddress	City	StateProvinceCode	CountryCode	ZipCode
1	26	Aaron	NULL	Barger	1941-12-16	3993 Lindale Avenue	Fremont	*M	US	94536
2	40	James	I	Catalano	1863-08-28	2410 Sunny Glen Lane	Cleveland	*M	[illegible]	44115
3	46	Steven	M	Olin	1976-11-22	1572 Alexander Drive	Denton	*M	US	76201

	CustomerID	FirstName	MiddleName	LastName	Birthdate	StreetAddress	City	StateProvinceCode	CountryCode	ZipCode
1	26	Testing 1 2 3	NULL	Barger	1941-12-16	3993 Lindale Avenue	Fremont	*M	US	94536
2	40	Testing 1 2 3	I	Catalano	1863-08-28	2410 Sunny Glen Lane	Cleveland	*M	[illegible]	44115
3	46	Testing 1 2 3	M	Olin	1976-11-22	1572 Alexander Drive	Denton	*M	US	76201

	CustomerID	FirstName	MiddleName	LastName	Birthdate	StreetAddress	City	StateProvinceCode	CountryCode	ZipCode
1	26	Aaron	NULL	Barger	1941-12-16	3993 Lindale Avenue	Fremont	*M	[illegible]	94536
2	40	James	I	Catalano	1863-08-28	2410 Sunny Glen Lane	Cleveland	*M	US	44115
3	46	Steven	M	Olin	1976-11-22	1572 Alexander Drive	Denton	*M	US	76201

Figure 104: Partial Results Sets

Save your query and close the query tab, leaving SSMS open for the next Try It.

The two result sets in Figure 104 show the data before, during, and after the transaction. As you can see, executing the ROLLBACK TRANSACTION statement restored the original state of the data. If you replace the ROLLBACK TRANSACTION statement with a COMMIT TRANSACTION statement, the changes made inside the transaction will be made permanent.

Insert

An INSERT statement is used to add one or more new rows of data to the database. An INSERT statement can accept literal strings or be populated by using the results of a SELECT statement.

When passing literal values, you can either specify the columns that will be populated or pass values for every column in the same order in which the columns are defined.

Syntax

```
INSERT
    [TOP (expression) [PERCENT]]
    [INTO]
    {table_or_view_name | rowset_function_limited}
{
    [(column_list)]
    [<OUTPUT_Clause>]
    {VALUES ({DEFAULT | NULL | expression} [,...n])
    |derived_table
    }
}
    |DEFAULT VALUES
```

To pass values for only a few columns, define the columns and order that you will be passing in data. This tells SQL Server how to interpret the data and which data goes in each field.

Copyright © 2018 Ann L. Weber

	No matter which method you use to populate the data for an INSERT statement, you must pass in a value for every column that does not allow NULL values and also does not have a default value defined to auto populate the column. Additionally, you cannot pass a value into a column defined as an IDENTITY column unless you first enable IDENTITY INSERT.

Sample – defined columns

```
BEGIN TRANSACTION
INSERT INTO Customer (CustomerID, FirstName, LastName,CountryCode)
             VALUES (500,        'Ann',     'Smith', 'US')
                  , (501,        'Bob',     'Jones', 'US')
                  , (502,        'John',    'Casey',  NULL)
;
```

When you are defining the columns that are being passed in, you can skip providing values for columns that have a default defined or that allow NULLs by specifying the key words DEFAULT or NULL in place of the data.

Sample – pass by position

```
BEGIN TRANSACTION
INSERT INTO Customer
  VALUES
  (503,'Ann',NULL,'Smith', NULL, NULL, NULL, NULL,'US', NULL)
 ,(504,'Bob',NULL,'Jones', NULL, NULL, NULL, NULL,'US', NULL)
 ,(505,'John','L','Casey', NULL, NULL, NULL, NULL,'US', NULL)
;
```

INSERT SELECT

INSERT SELECT is the name sometimes given to an INSERT statement that is populated with the results of a SELECT statement. If the data types in the results cannot be implicitly converted to those required by the destination table, you must explicitly convert them as part of the SELECT statement.

Sample

```
INSERT INTO SampleInsertSelect
SELECT C.CustomerID, C.FirstName, C.LastName, C.ZipCode
FROM Customer AS C
UNION
SELECT E.EmployeeID, E.FirstName, E.LastName, 'N/A'
FROM Employee AS E
;
```

INSERT SELECT vs SELECT INTO

An INSERT SELECT is demonstrated in the above example, and this statement inserts rows into any existing table based on data in one or more tables.

On the other hand, SELECT INTO statement creates a new table out of the results from a SELECT statement. Data types, column names, etc. are all based on the result set from the SELECT.

 Copyright © 2018 Ann L. Weber

If the table name in the SELECT INTO statement begins with one or two pound signs (#), the table is a temporary table and will be removed when the session where the table was created is closed. If a pound sign is not used when running a SELECT INTO statement, the user must have the CREATE TABLE permission in the destination database, and this creates a new permanent table. This option can be helpful when you want to analyze the data in a different way and play with "what if" scenarios.

Syntax

```
SELECT
    [TOP expression [PERCENT] [WITH TIES]]
    <select_list>
    [INTO new_table]
    [FROM {<table_source>} [,...n]]
    [WHERE <search_condition> ]
    [GROUP BY group_by_expression [,...n]
[HAVING <search_condition >]
```

Sample – SELECT INTO temporary table

```
SELECT C.FirstName, C.LastName
   , A.PrimaryCustomerID, A.OpeningBalance
   , AT.AccountType
INTO #TempSELECTINTO
FROM Customer AS C
   INNER JOIN Account AS A
   ON C.CustomerID = A.PrimaryCustomerID
   INNER JOIN AccountType AS AT
   ON A.AccountTypeID = AT.AccountTypeID
;
```

Sample – SELECT INTO new permanent table

```
SELECT C.FirstName, C.LastName
   , A.PrimaryCustomerID, A.OpeningBalance
   , AT.AccountType
INTO SELECTINTOSample
FROM Customer AS C
   INNER JOIN Account AS A
   ON C.CustomerID = A.PrimaryCustomerID
   INNER JOIN AccountType AS AT
   ON A.AccountTypeID = AT.AccountTypeID
;
```

Both samples above retrieve rows based on the join of three tables into a new table. The only difference between the two queries is the permanence of the tables. The first sample retrieves the data into a temporary table that will only last until you either close the session or execute a DROP TABLE statement. The second table is permanent in the database and will exist until you issue a DROP TABLE command.

<table>
<tr>
<td> </td>
<td>A single pound sign defines the temporary table as a “local” temporary table, accessible only from the session in which it is created. When two pound signs precede the name, the table is a “global” temporary table accessible to all connections that know of its existence. Be careful with global temporary tables because they still go away when the initial connection where the table was created is closed. If another connection is currently holding an active execution and lock on the table, the table will remain in memory until the execution is completed. Then SQL Server will automatically remove the table from the tempdb database.</td>
</tr>
</table>

Try It 2 – INSERT

In this Try It exercise you will create a new table named CustomerAccountExtended based off of a SELECT statement. The new table should have the customer first and last names together as a single column called CustomerName, along with the CustomerID, AccountNumber, AccountID, and OpeningDate columns. These columns will come from the Customer, Account, and CustomerAccount tables. Retrieving all columns from the new table should look similar to Figure 105.

	CustomerName	CustomerID	AccountNumber	AccountID	OpeningDate	OpeningDateOffset
1	Shaina Adams	2	SAdams1	1	2007-03-29	2007-03-29 00:00:00.0000000 +00:00
2	Peter Morgan	299	PMorga1	1	2007-03-29	2007-03-29 00:00:00.0000000 +00:00
3	Bonnie Spearman	3	BSpear2	2	2009-06-13	2009-06-13 00:00:00.0000000 +00:00
4	Abby Britt	109	ABritt2	2	2009-06-13	2009-06-13 00:00:00.0000000 +00:00
5	Shaina Adams	2	SAdams3	3	2006-04-13	2006-04-13 00:00:00.0000000 +00:00
6	Dorothy Smithwick	39	DSmith3	3	2006-04-13	2006-04-13 00:00:00.0000000 +00:00
7	Minnie Cayton	60	MCayto3	3	2006-04-13	2006-04-13 00:00:00.0000000 +00:00
8	Darwin Humphrey	260	DHumph3	3	2006-04-13	2006-04-13 00:00:00.0000000 +00:00

Figure 105: Partial Results Set

Note: You will add the OpeningDate twice, once converted to the datetimeoffset data type and aliased as OpeningDateOffset. You will use this column in the next Try It.

1. Add a new Query Editor tab and save your new query as **\Student Files\INSERT.sql**.
2. Set the database context to RetailBankingSample.
3. Write and execute the following query to retrieve the required columns from the Customer, Account, and CustomerAccount tables labeled as **CustomerName**, **CustomerID**, **AccountNumber**, **AccountID**, **OpeningDate**, and **OpeningDate** aliased as **OpeningDateOffset** and converted to the datetimeoffset data type. Only rows in common between all three tables should be returned. It doesn’t matter which table you retrieve the ID columns from. **459 rows** should be returned.

 Copyright © 2018 Ann L. Weber

```
SELECT  C.FirstName + ' ' +  C.LastName AS CustomerName
     , C.CustomerID
     , CA.AccountNumber
     , A.AccountID, A.OpeningDate
     , CONVERT(datetimeoffset, A.OpeningDate)
        AS OpeningDateOffset
FROM Customer AS C
     INNER JOIN CustomerAccount AS CA
          ON C.CustomerID = CA.CustomerID
     INNER JOIN Account AS A
          ON A.AccountID = CA.AccountID
ORDER BY C.CustomerID
;
```

4. Add the key word INTO followed by the new table name of CustomerAccountExtended. The INTO clause goes after the column listing in the SELECT clause and before the FROM clause as shown below.

```
SELECT  C.FirstName + ' ' +  C.LastName AS CustomerName
     , C.CustomerID
     , CA.AccountNumber
     , A.AccountID, A.OpeningDate
     , CONVERT(datetimeoffset, A.OpeningDate)
        AS OpeningDateOffset
INTO CustomerAccountExtended
FROM Customer AS C
     INNER JOIN CustomerAccount AS CA
          ON C.CustomerID = CA.CustomerID
     INNER JOIN Account AS A
          ON A.AccountID = CA.AccountID
ORDER BY C.CustomerID
;
```

5. Execute the query.
6. Write and execute a query to retrieve all rows and all columns from the CustomerAccountExtended table.
7. Write and execute the following query to insert a new row into the CustomerAccountExtended table with the following values:
    a. CustomerID – 500
    b. CustomerName – Your first and last name
    c. AccountNumber – Your first initial, the first 5 letters of your last name, then 900
    d. AccountID – 900
    e. OpeningDate – the current date

```
INSERT INTO CustomerAccountExtended
     VALUES ('Ann Weber',500, 'AWeber900', 900
             , GETDATE(), SYSDATETIMEOFFSET()
            )
;
```

8. Write a SELECT statement to review the new row that you inserted.
9. Save your query and close the query tab. Leave SSMS open for the next Try It exercise.

Update

The UPDATE statement is used to add, modify, or remove data in individual columns. Like the INSERT statement, the UPDATE statement accepts both literals and expressions based on SELECT statements as input. The WHERE clause defines which rows will be updated.

Syntax

```
UPDATE
    [TOP (expression) [PERCENT]]
    {table_or_view_name| rowset_function_limited}
    SET
       { column_name = {expression | DEFAULT | NULL}
       | column_name
         .WRITE (expression , @Offset , @Length)
        } [,...n]
    [<OUTPUT_Clause>]
    [FROM <table_source>]
    [WHERE <search_condition>]
```

The UPDATE statement includes five important sections:

1. The name of the table or view to be updated.
2. The column with the data to be updated. Both the column and the new value are defined as part of the SET clause.
3. The new value of the data. The new value can be the result of a SELECT statement or retrieved from another table referenced in the FROM clause.
4. A table source definition in the FROM clause can be added if either the new value or the WHERE clause references an additional table. The table from Step 1 and any additional tables will be joined together either through a JOIN or a correlated subquery.
5. The definition of what rows will be update. This restriction is defined in the WHERE clause. If the WHERE is left off, all rows in the table specified in the SET statement will be updated.

Sample with a fixed value

```
UPDATE SampleInsertSelect
SET Zipcode = '00000'
WHERE Zipcode = 'N/A'
;
```

The sample above changes the zip code value from N/A to 00000. The SampleInsertSelect table was created in an earlier sample by combining information in the Employee and Customer tables. This created a few concerns, including the formatting of the zip code, fixed above, and the duplicate

 Copyright © 2018 Ann L. Weber

PersonID values dealt with the in the query below. In the next sample, the PersonID value is incremented by 500 for all IDs where the PersonID, FirstName, and LastName fields all match. If the JOIN were to be performed only on the PersonID matching the EmployeeID, both customers and Employees with IDs less than and equal to 50 will be updated, which we don't want to happen.

Sample based on SELECT in both SET and WHERE clauses

```
UPDATE SampleInsertSelect
SET PersonID = E.EmployeeID + 500
FROM SampleInsertSelect AS S
LEFT OUTER JOIN Employee AS E
ON E.EmployeeID = S.PersonID
   AND E.LastName = S.LastName
   AND E.FirstName = S.FirstName
WHERE E.EmployeeID IS NOT NULL
;
```

	Don't forget the WHERE clause or every row in your table will be updated. If you also forget to enclose the UPDATE statement in an explicit transaction, manually typing the corrections or restoring from backup would be the only recourses for getting the original data back.

Try It 3 - UPDATE

In this exercise you will work through the logical steps to update the OpeningDateOffset column to the opening date with a time zone offset of -8 for all of the customers that live in CA.

Hint: You will use the SWITCHOFFSET function to update the values in the existing column.

1. Add a new Query Editor tab and save your new query as **\Student Files\UPDATE.sql**.

   Note: If you did not complete the previous Try It exercise, browse to **\Chapter 09 DML\Try It Exercises\ Try It 3 – UPDATE Starter.sql**. After you set the database to RetailBankingSample, execute the queries under Step #4 and Step #7 to create and populate the CustomerAccountExtended table.
2. Verify that the RetailBankingSample database is active.
3. Below the existing queries, write and execute a query similar to the one below that will retrieve the CustomerID, StateProvinceCode, and OpeningDate from the CustomerAccountExtended and Customer tables for Customers with a StateProvinceCode of 'CA'. The results should look similar to Figure 106. **35 rows** should be returned.

	CustomerID	StateProvinceCode	OpeningDate	OpeningDateOffset
1	127	CA	2015-05-31	2015-05-31 00:00:00.0000000 +00:00
2	243	CA	1985-01-30	1985-01-30 00:00:00.0000000 +00:00
3	127	CA	1994-03-10	1994-03-10 00:00:00.0000000 +00:00
4	127	CA	2011-02-24	2011-02-24 00:00:00.0000000 +00:00
5	257	CA	1992-01-26	1992-01-26 00:00:00.0000000 +00:00
6	222	CA	2008-04-27	2008-04-27 00:00:00.0000000 +00:00
7	234	CA	2010-02-19	2010-02-19 00:00:00.0000000 +00:00
8	20	CA	1991-01-02	1991-01-02 00:00:00.0000000 +00:00

Figure 106: Partial Results Set

```
SELECT C.CustomerID, C.StateProvinceCode
     , CE.OpeningDate, CE.OpeningDateOffset
FROM CustomerAccountExtended AS CE
     INNER JOIN Customer AS C
     ON C.CustomerID = CE.CustomerID
WHERE C.StateProvinceCode = 'CA'
;
```

4. Write and execute a query to verify that syntax of the SWITCHOFFSET command to modify the current offset of the California custom records to -8. Write your own query using the online help pages, or use the command below.

```
SELECT C.CustomerID, C.StateProvinceCode
     , CE.OpeningDate
     , SWITCHOFFSET(OpeningDateOffset, '-08:00')
FROM CustomerAccountExtended AS CE
     INNER JOIN Customer AS C
     ON C.CustomerID = CE.CustomerID
WHERE C.StateProvinceCode = 'CA'
;
```

5. Write and execute a BEGIN TRANSACTION statement.
6. Remove the SELECT clause from the query that you wrote in step 3 and replace it with the UPDATE and SET lines required to modify the OpeningDateOffset column for all customers in CA to UTC -8 as shown below.

```
SELECT C.CustomerID, C.StateProvinceCode
  , CE.OpeningDate, SWITCHOFFSET(OpeningDateOffset, '-08:00')
FROM CustomerAccountExtended AS CE
  INNER JOIN Customer AS C
  ON C.CustomerID = CE.CustomerID
WHERE C.StateProvinceCode = 'CA'
;
```

7. Write and execute a query that allows you to verify the results. Notice that the dates in the OpeningDateOffset column are now the previous day. This is because we built the new datetimeoffset column from a column with a date

 Copyright © 2018 Ann L. Weber

datatype. This caused every record to have a time of midnight UTC (offset 0). When we subtracted 8 hours, it moved the date to the previous day.

```
SELECT C.CustomerID, C.StateProvinceCode
     , CE.OpeningDate, CE.OpeningDateOffset
FROM CustomerAccountExtended AS CE
     INNER JOIN Customer AS C
     ON C.CustomerID = CE.CustomerID
WHERE C.StateProvinceCode = 'CA'
;
```

8. If the results are correct, issue a COMMIT statement to commit the transaction to the database. Otherwise, ROLLBACK and try again.
9. Save and close your query. Leave SSMS open for the next Try It exercise.

DELETE

DELETE statements are used to remove rows from a table. Like the UPDATE statement, the WHERE clause defines what rows will be deleted. If you forget the WHERE clause, every row in the table will be deleted. Unless you performed the delete inside of an explicit transaction that has not been committed, there is no way to undo or rollback the deletion.

Like INSERT or UPDATE statements, the WHERE clause can be defined as a fixed value or as a SELECT statement.

Syntax

```
DELETE
    [TOP (expression) [PERCENT]]
    [FROM]
    {table_or_view_name | rowset_function_limited}
    [OUTPUT <dml_select_list>]
    [FROM <table_source>[,...n]]
    [WHERE <search_condition>]

<dml_select_list> ::=
{<column_name> | scalar_expression}
  [[AS] column_alias_identifier]
    [,...n]

<column_name> ::=
{DELETED | INSERTED | from_table_name}.{* | column_name}
```

The DELETE statement can include two separate FROM statements. The first FROM clause defines the table name from which the rows will be removed. The word FROM is optional. The second FROM clause provides the same functionality as the FROM clause in an UPDATE statement. It is used to join additional tables from which information is retrieved to help define the WHERE clause and the rows that will be deleted.

The first sample below removes the row from the SampleInsertSelect table for the customer with an ID of 55.

Sample DELETE based on data in same table

```
BEGIN TRANSACTION
DELETE FROM SampleInsertSelect
WHERE PersonID = 55;
```

Sample DELETE based on data in another table

```
DELETE FROM SampleInsertSelect
FROM SampleInsertSelect AS SI
INNER JOIN Customer AS C
ON SI.PersonID = C.CustomerID
WHERE StateProvinceCode = 'OH'
;
```

The sample above removes all customers from Ohio from the SampleInsertSelect table. Eighteen rows are removed, leaving 332 rows in the table.

- Add a BEGIN TRANSACTION statement before performing any INSERT, UPDATE, or DELETE statements. Test the results before issuing a COMMIT statement.
- Write all UPDATE and DELETE statements first as a SELECT to test the rows that will be modified, then copy the WHERE and other relevant parts from the SELECT to the DML statement. This is also true for INSERT statements based off of the results of a SELECT statement.

TRUNCATE TABLE

In addition to being able to delete every row from a table by issuing a DELETE [FROM] *Table* command without a WHERE clause, you can also use a TRUNCATE TABLE statement. There are some benefits and limitations to this command.

Benefits

- Minimally logged operation.
  - DELETE must log every record, which can be cumbersome when removing all rows from large tables.
  - Takes less transaction log space.
  - Typically requires fewer locks
- Can still be rolled back when within an explicit transaction. The rows are only marked for deletion until the transaction is committed so that the modification can be rolled back.
- Resets any IDENTITY columns to their original seed value.
- Can remove rows from one or more partitions in partitioned tables.

 Copyright © 2018 Ann L. Weber

- All pages are deallocated. DELETE can leave empty pages behind.

Limitations

- Will not work on tables with active foreign key constraints that reference the table you want to empty. This is true even if there is not any data in the other table.
- Will not work on tables that participate in transactional or merge replication
- Will not work on tables that have an indexed view that is dependent on that table.

Syntax

```
TRUNCATE TABLE
    [ { database_name .[ schema_name ] . | schema_name . } ]
    table_name
    [ WITH ( PARTITIONS ( { <partition_number_expression> |
<range> }
    [ , ...n ] ) ) ]
[ ; ]
```

Sample

```
TRUNCATE TABLE SampleInsertSelect;
```

Try It 4 – DELETE

In this exercise you will explore the process of determining what rows will be deleted when performing a DELETE statement by writing a SELECT statement and then safely deleting the rows from within a transaction, verifying the results before committing the transaction.

1. Add a new Query Editor tab and save your new query as **\Student Files\UPDATE.sql**.

   Note: If you did not complete the Try It 1 exercise, browse to **\Chapter 09 DML\Try It Exercises\ Try It 3 – UPDATE Starter.sql**. After you set the database to RetailBankingSample, execute the queries under Step #4 and Step #7 to create and populate the CustomerAccountExtended table.
2. Verify that the RetailBankingSample database is active.
3. Within a transaction that you will roll back, write and execute a TRUNCATE TABLE command to empty the CustomerAccountExtended table as shown below.

```
BEGIN TRANSACTION;
TRUNCATE TABLE CustomerAccountExtended;
```

4. Write a SELECT statement to verify that the table is empty, and then ROLLBACK the transaction that you started in step 3. Rerun the SELECT to make sure that your data is back.

```
SELECT * FROM CustomerAccountExtended;
ROLLBACK;
```

Note: If you forgot the transaction and all of your data is permanently gone, rerun the queries in steps #4 and 7 in the \Chapter 09 DML\Try It Exercises\ Try It 3 – UPDATE Starter.sql file.

5. Write and execute a query that will return the rows in the CustomerAccountExtended table that have an OpeningBalance of 0 in the Account table. Because the requirement is to remove all customers for that account, not just the primary customer, you will only join based on the CustomerID as shown below. **51 rows** should be returned.

```
SELECT CE.CustomerID, CE.AccountNumber
FROM CustomerAccountExtended AS CE
     INNER JOIN Account AS A
     ON A.AccountID = CE.AccountID
     AND A.PrimaryCustomerID = CE.CustomerID
WHERE OpeningBalance = 0
;
```

6. Write and execute a query that will run inside of a transaction that will delete all Customers from the CustomerAccountExtended table where the OpeningBalance in the Account table is 0. **51 Rows** should be deleted.

```
BEGIN TRANSACTION;
DELETE FROM CustomerAccountExtended
FROM CustomerAccountExtended AS CE
     INNER JOIN Account AS A
     ON A.AccountID = CE.AccountID
     AND A.PrimaryCustomerID = CE.CustomerID
WHERE OpeningBalance = 0
;
```

7. Write a SELECT statement to verify the results and then COMMIT the transaction.

```
SELECT *
FROM CustomerAccountExtended AS CE
  INNER JOIN Account AS A
  ON A.AccountID = CE.AccountID
  AND A.PrimaryCustomerID = CE.CustomerID
WHERE OpeningBalance = 0
;
COMMIT;
```

8. Save and close your query. Leave SSMS open for the Lab.

 Copyright © 2018 Ann L. Weber

Chapter 10 - Data Definition Language

In this chapter:

Creating Tables
ALTER TABLE
DROP TABLE
Creating indexes
DROP INDEX
When to use indexes
Using the Graphical Execution Plan and Missing Index Hints
Chapter 10 Lab
Answers to Exercises

Files needed:

- \Chapter 10 DDL\Inline Samples
- \Chapter 10 DDL\Try It Exercises
- \Chapter 10 DDL\Labs\

	Some of the Try It exercises in this chapter build on one another, but are independent of other chapters. Answer files can be found in the \Chapter 10 DDL\Try It Exercises folder.

Creating Tables

During this course we have been dealing primarily with SELECT statements and data manipulation. Even though this is not a SQL Server Developer course, there are a few developer topics that are helpful to know, especially when working on analysis and quality assurance projects.

Creating tables is one such topic. The CREATE TABLE statement is an example of Data Definition Language (DDL). The three primary DDL commands are:

- CREATE – Defining a new object
- ALTER – Modifying an existing object
- DROP – Deleting an existing object

To be able to execute these commands, you must first have the appropriate permissions. Most users have permissions to create temporary tables, but in many organizations, only a few users have permission to create tables in the database. If you are working on your own local developer copy of SQL Server or have a "sandbox" environment to work in, you are more likely to have the required permissions.

	The full syntax, descriptions of each parameter, etc. are available in the SQL Server documentation at https://docs.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql?view=sql-server-2017.

Syntax – Simple Version

```
CREATE TABLE
    [ database_name . [ schema_name ] . | schema_name . ]
table_name
    (column_name <data_type> [ ,...n ] )
[ ; ]
```

Sample

```
CREATE TABLE NewUsers
  (UserID int IDENTITY (1,1) PRIMARY KEY NOT NULL
       , FirstName varchar(20) NOT NULL
       , MiddleName varchar(20) NULL
       , LastName varchar(20) NOT NULL
  )
;
```

As seen above, when creating a new table, you must specify the table name and at least one column definition including the column name and data type. Selecting the correct data type for each column is extremely important.

Below are just a few examples of places where choosing a different data type can make a significant difference:

 Copyright © 2018 Ann L. Weber

- When working with financial data, the currency data type goes to four decimal places. If the work you are performing requires consistent rounding to 2 or 6 decimal places, decimal is a better data type selection.
- When working with large datasets, having a primary key of an Integer data type will become a problem when your dataset exceeds 2 million rows.
- When storing numbers, using a character data type will change how the numbers are displayed and sorted. Leading 0's won't be lost, but 2 will come after 10.
- Storing dates as strings will not allow you to easily manipulate the dates using the many built-in functions.

In each of these and many other situations, there is not always one "right" answer. Be aware of what can happen, analyze your requirements, and make the best decision based on your data, requirements, and business rules.

See Chapter 3 Built-in Functions Overview for additional information on common data types and their functionality. Additionally, see Chapter 8 Importing Data for information for warnings about how data types affect data imports.

Additional features such as primary keys, whether or not the columns support NULL values, identity columns and more are all optional settings when creating a table. The IDENTITY key word creates a column that is populated automatically when data is entered. The value will start at the seed value (the first number in the parentheses) and then increment by the second number.

Constraints

Although a full discussion of table and referential constraints is beyond the scope of this course, the list below provides a brief description of the types of constraints available in SQL Server.

- **PRIMARY KEY** – one or more columns defined to uniquely identify each row in a table. PRIMARY KEY constraint columns cannot allow NULL values. When the key is made up of more than one column, it is referred to as a composite key. The PRIMARY KEY constraint for the CustomerAccount table in the RetailBankingSample database is an example of a composite key. There can only be one PRIMARY KEY constraint per table.
- **UNIQUE** – like a primary key, a UNIQUE constraint requires every row to have a unique value in that column, but, it will allow a single NULL value. Because you can only have one primary key field per table, UNIQUE constraints can be used for fields that, due to size or other limitations, do not make good Primary Key fields, but still should be unique. Examples of columns that would use a UNIQUE constraint are automobile VIN numbers, Social Security numbers, and product serial numbers.
- **FOREIGN KEY** – one or more columns used to create a relationship between two tables to support referential integrity. For example, a FOREIGN KEY

constraint added to the Account table on the PrimaryCustomerID column that references the CustomerID in the Customer table will prohibit records from being added to Account table unless they have a valid CustomerID value in the Customer table.

- **CHECK** – defines acceptable values for a column that are more restrictive than the data type restrictions. You can use CHECK constraints to make sure that an employee's hire date is greater than their birthdate or at least 16 years after their birthdate. When comparing values in different columns, all columns must exist in the same table.
- DEFAULT – defines a column value to be entered automatically when a new row is added and no value is defined for the column.

Try It 1 – CREATE TABLE

In this exercise, you will create a new table called MyTransactions. This table will be used throughout the Try It exercises in this chapter. The table will include a column called NewTranKey. This column will be a primary key column and should auto-increment starting with the number 1, incrementing by 1, and use the integer data type. The other columns are defined as follows:

- BusinessTranKey int
- AcctID int
- Amount numeric (16,6)
- TransactionType
- TransactionDate date

1. Open a new query window and save the query as **CREATE.sql** to \Student Files.
2. Set the database context to **RetailBankingSample**.
3. Type and execute the following command to create the **MyTransactions** table as defined at the beginning of this Try It exercise.

```
CREATE TABLE MyTransactions
     (NewTranKey int PRIMARY KEY IDENTITY (1,1)
      , BusinessTranKey int
      , AcctID int
      , Amount numeric (16,6)
      , TransactionType varchar(20)
      , TransactionDate date
     )
;
```

4. Click **File | Open File**, browse to **\Chapter 10 DDL\Try It Exercises**, and then open the **Try It 1 – CREATE Populate.sql** file.
5. Review and execute the script to populate the **MyTransactions** table.
6. Save and close the queries, but leave SSMS open for the next Try It.

 Copyright © 2018 Ann L. Weber

ALTER TABLE

After creating a new table, the ALTER TABLE command allows you to make changes to the table structure. You can add additional columns, change the names of existing columns, increase the size of a data type in a column, and more. If you drop a column, all data in that column will be lost. Additionally, if you change a column data type to a smaller or less accurate data type, you may lose accuracy or SQL Server may truncate the data.

Syntax

```
ALTER TABLE [schema_name.] table_name
{
    ALTER COLUMN column_name
    {
        [type_schema_name.] type_name
        [({precision[, scale]|max})]
        [NULL|NOT NULL]
    }
    |[WITH {CHECK|NOCHECK}] ADD
    {
        <column_definition>
      | <computed_column_definition>
      | <table_constraint>
    } [,...n]
    | DROP
    {
        [CONSTRAINT] constraint_name
        | COLUMN column_name
    } [,...n]
    | [WITH {CHECK|NOCHECK}] {CHECK|NOCHECK} CONSTRAINT
        { ALL|constraint_name [,...n]}
  | ADD COLUMN column_name data_type [additional options]

};
```

The syntax of this command can be quite complex with numerous options that are beyond the scope of this course. You can read more about the full syntax of ALTER TABLE at https://docs.microsoft.com/en-us/sql/t-sql/statements/alter-table-transact-sql?view=sql-server-2017.

Sample

```
ALTER TABLE NewUsers
  ALTER COLUMN FirstName varchar(50)
;

ALTER TABLE NewUsers
  ADD Title varchar(20) NULL
;
```

Try It 2 – ALTER TABLE

In this exercise, you will add a new column named ImportedDate with a data type of datetime2 to the MyTransactions table that you created in the previous Try It. You will also change the data type of the TransactionDate column to datetimeoffset.

1. Open a new query window and save the query as **ALTER.sql** to **\Student Files**.
2. Set the database context to **RetailBankingSample**.
3. Type and execute the following command to add a new column named **ImportedDate** with a data type **datetime2**. Use a transaction so that you can roll back the command if anything goes wrong. Remove the comment marks to run the COMMIT statement.

```
BEGIN TRANSACTION;

ALTER TABLE MyTransactions
     ADD ImportedDate datetime2
;

--COMMIT;
```

4. Use a transaction to add **the current date and time** as a **datetime2** data type into the **InsertedDate** column in every row. Once you have verified the ALTER TABLE executed successfully, remove the comment marks to run the COMMIT statement.

```
BEGIN TRANSACTION
UPDATE MyTransactions
     SET ImportedDate = SYSDATETIME()
;

SELECT * FROM MyTransactions;
--COMMIT;
```

5. Type and execute the following command to change the data type on the **TransactionDate** column to **datetimeoffset**.

```
ALTER TABLE MyTransactions
ALTER COLUMN TransactionDate datetimeoffset
;
```

6. Save and close the ALTER.sql file. Leave SSMS open for the next Try It exercise.

 Copyright © 2018 Ann L. Weber

DROP TABLE

The DROP TABLE statement removes the table completely from the database. This includes both the table definition and any data that had been loaded into the table.

Syntax

```
DROP TABLE [ IF EXISTS ] [ database_name . [ schema_name ] . |
schema_name . ]
table_name [ ,...n ]
[ ; ]
```

The IF EXISTS optional phrase only applies to SQL 2016 and later. IF EXISTS conditionally drops the table if it already exists. This helps to avoid error messages when trying to remove a table that was either already removed or not yet created as part of an automated script. In early versions, programmers would write an IF statement to test to see if the table existed in the sys.objects system view before issuing the DROP TABLE command. This new feature makes the process much easier.

Sample

```
DROP TABLE NewUsers;
```

	You should always exercise great care when issuing a DROP TABLE command. Unless you perform the DROP TABLE statement inside of a transaction, there is no way, other than restoring a backup, to recover the data in the table.

Try It 3 – DROP TABLE

In this exercise you will use a transaction to test dropping the MyTransactions table from the previous exercises. You will then roll back the transaction so that the table will be available for later Try It exercises.

1. Open a new query window and save the query as **DROP.sql** to **\Student Files**.
2. Set the database context to **RetailBankingSample**.
3. Write and execute the following code to start a transaction and then drop the MyTransactions table.

   ```
   BEGIN TRANSACTION
   DROP TABLE MyTransactions;
   ```

4. Write and execute a SELECT statement against the **MyTransactions** table to verify that the table is gone.
5. Write and execute a ROLLBACK statement as shown below.

```
ROLLBACK;
```

6. Rerun the SELECT statement from step 4 to verify that the table and data are both intact and available.
7. Save and close the query. Leave SSMS open for the next exercise.

Creating indexes

Although a full discussion on indexes and performance tuning is outside of the scope of this class, it is helpful to understand the basics of creating indexes, especially when you are working in an environment where you are creating your own tables and/or working with large datasets. SQL Server includes two general types of indexes: clustered and non-clustered indexes.

Non-clustered indexes

A non-clustered index is similar to the index at the back of the book. The index is in a separate location on disk that holds references to the actual data. These references are sorted on one or more columns that are defined at the time the index is created.

Non-Clustered indexes are the default index type when you execute a CREATE INDEX statement on a table.

Included Columns

The included columns feature was added to non-clustered indexes in SQL Server 2005. These non-key columns are added to the leaf-level of the non-clustered index and can enhance performance by "covering queries". A covered query retrieves all rows for the query by using the non-clustered index and not having to access the physical table.

Filtered Indexes

The ability to filter an index was added in SQL Server 2008. This feature allows you to limit the number of rows in an index based on a WHERE clause. This feature can greatly boost performance on extremely large tables where only a small portion of rows are regularly returned. The WHERE clause of the SELECT statement must match the WHERE clause of the index definition.

Syntax

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX
    index_name
    ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WHERE <filter_predicate> ]
 [ ; ]
```

Sample

```
CREATE NONCLUSTERED INDEX
  nc_AccountTransaction_AcctID_2017
  ON AccountTransaction (AcctID)
  INCLUDE (AMOUNT)
```

 Copyright © 2018 Ann L. Weber

```
    WHERE TransactionDate > '20180101'
;
```

If you are writing queries that join the Account table to the AccountTransaction table and want to return the Amount column from the AccountTransaction table for the current year along with other columns from the Account table or other tables, the index above will limit the number of pages that must be read to support the query. This sample uses the year 2018 as the current year. If your queries regularly return different years, but only one year at a time, then you should include Transaction date in the index key instead of using a filtered index.

	I was teaching a performance tuning class for a company that dealt with stocks and indexes. On the last day of class we worked together to apply what they learned to their data. They had one extremely large table where only about 30% of the data was actively being queried. The rest was for historical purposes, but could not be moved to another location. There were about 20 columns in the table, and they needed to return 15 columns in most of the queries. Luckily, there was a bit field that was 1 for the 30% of the data that was needed and a 0 for the rest of the data. At the time, included columns and filtered indexes were fairly new, so I was unsure if the optimizer was going to like what we were doing, but we decided to try. We created an index on the foreign key field that was frequently used to connect to other tables. We included all 15 columns that were regularly used in queries, and most importantly, we filtered on our bit column being equal to 1. The optimizer used the new index for every query we tested. The most remarkable difference was that an 8-minute query went down to less than 30 seconds! On the down side, index maintenance was increased for the database administrators. Also, inserts and updates to that table were slowed slightly due to the index needing to be updated as well. The company felt the tradeoff was well worth it.

Try It 4– Create a non-clustered index

In this exercise, you will add an index to optimize a query that you will be running multiple times. The query has a WHERE clause that returns only one transaction type at a time. Additionally, the only columns being returned are the NewTranKey and Amount columns.

1. Open a new query window and save the query as **INDEX.sql** to **\Student Files**.
2. Set the database context to **RetailBankingSample**.
3. Type the following commands to enable statistics for both disk IO and CPU time.

```
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
```

4. Click the **Include Actual Execution Plan** icon.
5. Write and execute the following SELECT statement to return the **NewTranKey** and **Amount** columns from the **MyTransactions** table for transactions with a **TransactionType of Interest**.

```
SELECT MT.NewTranKey, MT.Amount
FROM MyTransactions AS MT
WHERE TransactionType = 'Interest'
;
```

6. Make note of the number of reads and execution time. **17,894 rows** are returned and the IO statistics should show around **2,746 logical** reads. Review the graphical execution plan. Notice that a Clustered Index Scan was used on the MyTransaction table. A scan means the whole table (index) is read in order.
7. Type and execute the following statement to create a non-clustered index to support the query defined in the Try It introduction.

 Copyright © 2018 Ann L. Weber

```
CREATE INDEX nc_MyTrans_TranType
     ON MyTransactions (TransactionType)
     INCLUDE (Amount)
;
```

8. Rerun the SELECT query. Notice that the logical reads is now around 101 and an index seek was used on the new index.

	Due to differences based on hardware specifications, SQL Server versions, database modifications, and more, the logical reads in these steps may vary from the numbers specified, but they should always be significantly lower in step 8.

Clustered Indexes

A clustered index is more like an encyclopedia. Even though this analogy is a bit outdated, it best describes how a clustered index works. There are non-leaf level pages (like the index book in a large encyclopedia set) that help the server quickly work through the index level to go straight to the area (or correct book in our encyclopedia analogy) where the actual data is stored in sorted order by the key column(s) used to define the key. This key is called the "Clustering Key," and the level of the index that holds the sorted data is called the "Leaf Level" of the index. Because the actual table data is stored in the leaf level of the clustered index, you can only have one clustered index per table.

Even though you can only have one clustered index per table, the clustered index can be created on more than one column. For example, if most of your queries are sorted by State code and Zip code, it may be beneficial to create the clustered index on these two fields rather than the typical clustered index of the primary key field for potentially better performance. Be careful though - the clustered index key is a part of every record in the non-leaf level of not only the clustered index but also every non-clustered index. If you have too large of a data field or too many columns in your clustering key, performance can be adversely affected.

When you create a PRIMARY KEY constraint on one or more columns in a table, by default a clustered index will be created to support the PRIMARY KEY constraint. If you already have a clustered index on another column or columns, SQL Server will create a non-clustered index on the primary key.

When a clustered index exists on a table, the leaf level of the index replaces the table. Data in a table without a clustered index is stored in what is referred to as a Heap.

DROP INDEX

Once you no longer need an index, you can remove the index without affecting the table. The DROP INDEX statement performs this functionality.

Syntax

```
DROP INDEX [ IF EXISTS ]  index_name ON <object>  ;
```

Copyright © 2018 Ann L. Weber

Sample

```
DROP INDEX nc_AccountTransaction_AcctID_2017
   ON AccountTransaction
;
```

When to use indexes

Although a full discussion of index optimization is beyond the scope of this course, in general, non-clustered indexes are best for retrieving small groups of rows based on a WHERE clause in your queries.

Clustered indexes on the other hand are optimal for returning sorted data or large ranges of rows.

If you have large tables, indexes on the columns used in JOIN ON clauses and those used in WHERE clauses can frequently improve performance.

Using the Graphical Execution Plan and Missing Index Hints

As you saw in Chapter 6 Joining Multiple Tables, you can use the SET STATISTICS options and the graphical execution plan to see various performance statistics, including:

- How much data is being retrieved and passed through the query steps
- What types of joins are being used
- How many pages are read in each table
- Is data being retrieved from the physical disks

Additionally, if a query would perform significantly better with an index, the server displays a "missing index hint" when you use the graphical execution plan. You can right-click these hints and see the SQL code for the CREATE INDEX statement between block comment symbols. Although the missing index hints can be beneficial, you should not automatically create each index. Rather, you should analyze existing indexes to see if they could be modified to fit the same need. Also, evaluate if the query is run frequently enough to warrant the overhead associated with an index.

	Be sure to change the name of the index from **<Name of Missing Index, sysname,>** to something meaningful.

Try It 5 – Using Missing Index hints.

In this exercise, you will explore creating an index based on the missing index hint for one of the subqueries written in Chapter 7.

 Copyright © 2018 Ann L. Weber

1. Open the **Try It 5 – Index Hints starter.sql** file from the **\Chapter 10 DDL\Try It Exercises** folder.
2. Highlight and execute the **USE RetailBanking** statement.
3. Click the Include Actual Execution Plan icon.
4. Highlight and execute the query in the script file.
5. Right-click the green Missing Index hint, and then click Missing Index Details.
6. Change the name of new index to nc_AccountTran_TranDate.
7. Highlight the USE Database and CREATE INDEX commands between the comment marks and execute the query.
8. Save the missing index script to the **\Student Files** folder as **Try It 5 – Index Hints Missing Index.sql**.
9. Return to the **Try It 5 – Index Hints starter.sql** tab and rerun the SELECT statement. Review the Graphical Execution plan to see if the query used the new index.
10. Close both queries and leave SSMS open for the lab.

Chapter 11 - Working with Temporary Objects

In this chapter:

Declaring variables
Importance of using correct data types
Table variables
Temporary Tables
Common Table Expressions (CTEs – If time permits)
Chapter 11 Lab
Answers to Exercises

Files needed:

- \Chapter 11 Temp Objects\Inline Samples
- \Chapter 11 Temp Objects \Try It Exercises
- \Chapter 11 Temp Objects \Labs\

	Answer files for the Try It Exercises can be found in the \ Chapter 11 Temp Objects \Try It Exercises folder.

 Copyright © 2018 Ann L. Weber

Declaring variables

Variables act as placeholders for data, making it easier to use different values each time you run a query. When you define a variable, you must provide a name and data type. Optionally you can also set the value of the variable when you declare it.

Variable names always start with an "at" symbol (@). The variable is available until the end of the current batch.

Syntax

```
DECLARE
{
    { @local_variable [AS] data_type  [ = value ] }
  | { @cursor_variable_name CURSOR }
} [,...n]
;
```

Sample

```
DECLARE @charactervar varchar(30) = 'testing 1 2 3 '
   , @numbervar int = 123
;
```

The example above creates and populates two separate variables that can then be used within the same batch where they are created. Variables only last for the duration of the batch execution, meaning, if you run the query using the variable a second time, you will need to run the DECLARE statement again as well.

In addition to setting the variable immediately as part of the DECLARE line, you can use either a SET or SELECT statement to populate the value of the variable. A SELECT statement typically returns a result set, except when used to set a variable. Rather, the SELECT simply sets the value of the variable and returns nothing as a result set. Using the SET statement rather than the SELECT statement causes less confusion.

Sample – SET

```
SET @charactervar = 'testing 4 5 6';
```

Sample – SELECT

```
SELECT @charactervar = 'testing 7, 8, 9';
```

You can also use the output of a SELECT statement to populate a variable.

Sample – Query Result

```
SET @charactervar = (SELECT MAX(FirstName) FROM Customer);
```

The prior samples all require the DECLARE statement to be executed as part of the batch. None of these statements will return a result set.

Try It 1 – Using Variables

In this exercise, you will create a variable to retrieve the most recent year in the OpeningDate column from the Account table. You will then use that year to retrieve the AccountTransaction rows that occurred during that year.

1. Open a new query tab and save the query to the **\Student Files** folder as **Variables.sql**.
2. Set the database context to RetailBankingSample.
3. Write the following statement to declare a variable named @intYear with a data type of int and set the value to the maximum OpeningDate in the Account table.

   ```
   DECLARE @intYear int = (SELECT YEAR(MAX(OpeningDate))
                           FROM Account);
   ```

4. Below the variable definitions, write a SELECT statement to return all columns from the AccountTransaction table where the TransactionDate is in the same year as the @intYear variable as shown below. **8,179 rows** should be returned.

   ```
   SELECT * FROM AccountTransaction
   WHERE YEAR(TransactionDate) = @intYear
   ;
   ```

5. Save and close the query. Leave SSMS open.

Importance of using correct data types

As you have seen throughout the class, it is important to choose the correct data type for your data. Due to the implicit conversion rules SQL Server employs when combining variables and columns of different data types within expressions, a poorly chosen data type can cause data to lose accuracy or to be truncated.

Try It 2 – Variable Data Types

In this exercise, you will view some of the ways data types affect variable usage. Although these are exaggerated samples, they display what happens when data types are poorly chosen.

1. 1. Open the **Try It 2 – Variable Data Types Starter.sql** file and then save it to the **\Student Files** folder as **VariableDataTypes.sql**.
2. Execute the USE RetailBankingSample; command.
3. Review the commands under --Step #2. Notice that variable and the CONVERT statement define different character limits.
4. Execute just the SELECT statement (without the parentheses) that is on the right side of the SET statement.

 Copyright © 2018 Ann L. Weber

5. Notice how the date is formatted in the result set.
6. Execute the full set of statements under --Step #2. Review the results, and notice that the last character was truncated without warning.
7. Review the queries under --Step #7. Notice that the extra spaces stored in the database are trimmed off of the first name field.
8. Execute the statements under --Step #7. Notice the difference between the char first name and the varchar first name.
9. Close the query window but leave SSMS open.

Table variables

Like local variables, table variables exist with the context of a batch. Unlike temp tables, table variables must be defined and then populated as two separate steps.

	For a more information on table variables and their pros and cons, see https://docs.microsoft.com/en-us/sql/t-sql/data-types/table-transact-sql?view=sql-server-2017 and https://blogs.msdn.microsoft.com/sqlserverstorageengine/2008/03/30/tempdb-table-variable-vs-local-temporary-table/.

Syntax

```
DECLARE
{
  { @local_variable [AS] TABLE ( { <column_definition> |
                               <table_constraint> } [ ,...n ] );
```

Sample

```
DECLARE @mytable table (Mykey int identity
                        , FirstName varchar(20)
                        , LastName varchar(20)
                        , CustomerKey int);
```

Try It 3 – Table Variables

In this Try It exercise you will use a table variable to store total amount values from the AccountTransaction table grouped by the CustomerID and the Year of the transaction. The resulting table variable will return the following columns and data types:

- AccountID - int
- TransactionYear - int
- TotalAmount - money

	AccountID	TransactionYear	TotalAmt
1	2	2009	-11375.83
2	2	2010	-46000.53
3	2	2011	-10950.98
4	2	2012	23336.89
5	2	2013	-11650.87
6	2	2014	-32293.69
7	2	2015	-3694.12
8	2	2016	23383.42

Figure 107: Temp Table Results Set

1. Open a new query tab and save the query to the **\Student Files** folder as **TableVariable.sql**.
2. Set the database context to RetailBankingSample.
3. Write the following statement to declare a table variable named @TransactionTotals with the columns and data types defined in the Try It introduction:

```
DECLARE @TransactionTotals table (AccountID int,
TransactionYear int, TotalAmt money);
```

4. You can execute the entire script now, but other than verifying that there aren't any runtime errors, you will not see results until you populate and then query the table variable.
5. Below the existing statements, type the following code to populate the table variable. Again, you can execute the entire script to verify that there are not any runtime errors, but you will not see results other than on the message tab that **987 rows** were affected.

```
INSERT INTO @TransactionTotals
SELECT AcctID, YEAR(TransactionDate) AS TransactionYear
     , SUM(Amount) AS Total
FROM AccountTransaction
GROUP BY AcctID, YEAR(TransactionDate)
```

 Copyright © 2018 Ann L. Weber

```
;
```

6. Below the existing statements, write a query to return all columns and all rows from the table variable sorted by AccountID then year, and then execute the entire script. Confirm 987 rows were returned.

```
SELECT * FROM @TransactionTotals
ORDER BY AccountID, TransactionYear;
```

7. Save and close the query tab, but leave SSMS open.

Temporary Tables

There are two general categories of temporary tables: local and global. Local temporary objects start with a single pound sign (#), global temporary tables begin with two pound signs (##).

Local temporary tables are only accessible to the session in which they are created. Global temporary tables are available to all sessions that are open at the same time. Temporary tables only persist for as long as the session where they were created remains open, unless they are manually dropped by using the DROP TABLE command. If a user in another connection uses a transaction that is holding one or more locks on the temporary table at the time the connection where the temporary table was created is closed, the table remains active in the tempdb database until the locks are released. As soon as the locks are released, SQL Server drops the table.

Like permanent tables, temporary tables can be created with either a CREATE TABLE command or a SELECT INTO statement. Temporary tables also support the creation of indexes.

Sample

```
SELECT RTRIM(FirstName) AS FirstName
       , RTRIM(LastName) AS LastName
       , CustomerID
INTO #TempCustomer
FROM Customer;
```

Try It 4 - Temporary Table

In this exercise you will create a temporary table based on the results of a query that joins the Customer and CustomerAccount tables. You will then attempt to run the query that created the temp table again. Finally, you will retrieve the data from the temp table.

1. Open a new query tab and save the query to the **\Student Files** folder as **TempTable.sql**.
2. Set the database context to RetailBankingSample.
3. Type and execute the following statement to create the **#TempCustomerAccountInfo** table.

```
SELECT C.FirstName, C.LastName, CA.AccountNumber
INTO #TempCustomerAccountInfo
FROM Customer AS C
     INNER JOIN CustomerAccount AS CA
     ON C.CustomerID = CA.CustomerID
;
```

4. Try to run the statement from step 3 again. Notice the error saying that the table already exists. However, you would be able to copy this command into a new query window and execute it. This is because the table is only seen in the local session and opening a new query window creates a new connection to the server.
5. Write and execute the following query to retrieve all of the columns and rows from the temporary table.

```
SELECT * FROM #TempCustomerAccountInfo;
```

6. Save and close the query tab. Leave SSMS open.

Common Table Expressions (CTEs – If time permits)

Common Table Expressions, frequently referred to as CTEs, are also temporary objects. The one major benefit of CTEs are their ability to be built recursively and referenced multiple times within a single query statement.

Like a table variable, the CTE is very short lived. In fact, CTEs only lasts for the duration of a single statement, while a table variable lasts for the duration of the executing batch.

Although CTEs are generally considered an advanced query technique, a sample with a brief description is included here to get you started.

Best Practice!	A CTE was the first command to require the previous statement to end in a semi-colon. Because of this, many programmers started putting a semi-colon (;) at the beginning of the CTE command. It is a much better practice to end every statement with a semi-colon and NOT start a CTE with one. More commands are being added with each version of SQL Server that require the semi-colon. Additionally, the Microsoft documentation states that in a future version, semi-colons will be required on all statements.

Syntax – Simple CTE

```
[ WITH <common_table_expression> [ ,...n ] ]

<common_table_expression>::=
    expression_name [ ( column_name [ ,...n ] ) ]
```

 Copyright © 2018 Ann L. Weber

```
AS
( CTE_query_definition )
```

With what Microsoft calls a "simple" CTE, the data is derived from a single pass through a SELECT statement. The SELECT statement can be as simple or complex as necessary. The results are then available to be queried while the CTE is built, as shown in the sample below.

Sample – Simple CTE – list of customers and how many accounts they opened each year

```
WITH Account_CTE (CustomerID, AccountID, OpeningYear)
AS
( SELECT C.CustomerID, A.AccountID, YEAR(A.OpeningDate)
   FROM Customer AS C
        INNER JOIN Account AS A
        ON C.CustomerID = A.PrimaryCustomerID
)
SELECT CustomerID, OpeningYear, COUNT(AccountID) AS
AnnualAccountOpening
FROM Account_CTE
GROUP BY OpeningYear, CustomerID
ORDER BY AnnualAccountOpening DESC
;
```

Sample – Recursive CTE – employees and their managers

```
WITH ReportsTo_CTE (EmployeeID, ManagerID, Title,
EmployeeLevel)
AS
( SELECT E.EmployeeID, E.ReportsTo, E.Title
        , 0 AS EmployeeLevel
   FROM Employee AS E
   WHERE ReportsTo IS NULL
   UNION ALL
   SELECT E.EmployeeID, E.ReportsTo, E.Title
         , EmployeeLevel + 1
   FROM Employee AS E
  INNER JOIN ReportsTo_CTE AS M
  ON E.ReportsTo = M.EmployeeID
)
SELECT EmployeeID, ManagerID, Title, EmployeeLevel
FROM ReportsTo_CTE
ORDER BY EmployeeLevel;
```

Recursive CTEs, like the sample above, use the UNION ALL statement to combine the first query results (referred to as the "anchor" rows) to the results of each recursive pass of the second query. In the sample above, the anchor query retrieves the employees who do not have a value in the ReportsTo column and sets their EmployeeLevel to 0. The server then takes the EmployeeID for each of the rows in the anchor results and runs the 2nd query to find the employees who report to the employees from the anchor row. Those employees will have a level of 1 (0 + 1). The server will then repeat the 2nd query for each new employee that is added to the CTE results, finding any employees who report to the newly added records. This process continues until no rows are returned based on the inner join.

 Copyright © 2018 Ann L. Weber