Transact-SQL

The Building Blocks to SQL Server Programming

By Gregory A. Larsen

Transact-SQL

The Building Blocks to SQL Server Programming

By Gregory A. Larsen

Edited by Kathi Kellenberger

Published by Redgate

First Edition 2020

For further expert content visit www.redgate.com/Hub

Copyright Gregory A. Larsen

The right of Gregory A. Larsen to be identified as the authors of this book has been asserted by them in accordance with the Copyright, Designs and Patents Act 1988. All rights reserved.

No part of this publication may be reproduced, stored or introduced into a retrieval system, or transmitted, in any form, or by any means (electronic, mechanical, photocopying, recording or otherwise) without the prior written consent of the publisher.

Any person who does any unauthorized act in relation to this publication may be liable to criminal prosecution and civil claims for damages. This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, re-sold, hired out, or otherwise circulated without the publisher's prior consent in any form other than which it is published and without a similar condition including this condition being imposed on the subsequent publisher.

Written by Gregory A. Larsen

Edited by Kathi Kellenberger

Published by Redgate

Contents

About the Author 1

Introduction 1

Chapter 1 - The Basic SELECT Statement 2

The Three Parts in a SELECT Statement 2

The Column List 3

The FROM Clause 5

The WHERE Clause 7

Business Cases 9

Exercise #1: 9

Exercise #2: 10

Answers to Exercise #1: 10

Answers to Exercise #2: 11

Understanding the Basic SELECT Statement 13

Chapter 2 - History of Structured Query Language (SQL) 14

The 1970s 14

The 1980s 15

The 1990s 16

The 2000s 17

The 2010s 17

The Rest is History 19

Chapter 3 - Implementing a Relational Model in SQL Server 20
The Father of Relational Data Modeling 20
Implementing a Relational Model in SQL Server 21
Validating Database Design 29
Relational Database Design 32
Chapter 4 - The Mathematics of SQL: Part 1 33
What is a table? 33
Basic Set Theory 33
Venn Diagrams 34
Mathematical Operators 37
EXCEPT operator 43
Joining Tables that Contain More than a Single Column 45
Set Theory and SQL Server 47
Chapter 5 - The Mathematics of SQL: Part 2 48
Joining Two Sets 48
Creating Sample Data for [SET A] and Set B 50
INNER JOIN Operator 52
OUTER JOIN Operator 53
Joining two sets with Different Number of Columns 56
Set Theory and the Mathematics of Joining SQL Server Tables 58

Chapter 6 - The Basics of Sorting Data Using the ORDER BY Clause ... 59
Ordering your records ... 59
Ordering by a Single Column ... 62
Sorting Data based on collation ... 63
Sorting Data in Descending Sequence ... 64
Sorting Data Based on Column Not in the SELECT List ... 66
Sorting Data based on Column Ordinal Position ... 66
Sorting Data based on Column Alias ... 67
Sorting Data Based on Multiple Columns ... 69
Numeric Character Data Sorted Alphabetically ... 69
Create Order to Your Selected Data ... 71
Chapter 7 - Using Logical Operators ... 72
Logical Operators ... 72
BETWEEN operator ... 73
LIKE Operator ... 75
IN Operator ... 78
Pitfalls of Using Logical Operators ... 81
Using Logical Operators ... 83
Chapter 8 - Using Scalar Functions ... 84
Categories of Scalar Functions ... 84
Determinism of a Function ... 85
Database Used for Examples ... 87
Examples of using Scalar functions ... 87

Date/Time Function Examples....87
String Function Examples....94
Conversion Function Example....96
Using Scalar Function to Support Application Requirements....98

Chapter 9 - Summarizing Data Using a Simple GROUP BY Clause....99

Simple GROUP BY clause....99
Sample Data for Exploring the Simple GROUP BY Clause....100
Grouping by a Single Column....102
Grouping by Multiple Columns....103
Using an Expression in the GROUP BY Clause....105
HAVING Clause....106
Summarizing Data with the Simple GROUP BY Clause....108

Chapter 10 - Using the ROLLUP, CUBE and GROUPING SET operator in a GROUP BY Clause....109

When are the ROLLUP, CUBE, and GROUPING SETS operators useful?....109
Sample Data....110
Using the ROLLUP operator to Create Subtotals and a Grand Total....112
Using the CUBE Operator to create a Superset of Aggregated Values....118
Creating Multiple Aggregated Groupings using the GROUPING SETS Operator....123
More than One Way to Aggregate Data: Using ROLLUP, CUBE and GROUPING SETS....127

Chapter 11 - Adding Records to a table using INSERT Statement....128

The Basic INSERT statement....128
Inserting Data into a Table using a SELECT statement....131

Inserting Data into a Table using a Stored Procedure 132
Using the OUTPUT Clause..... 133
Populating Data in a Table 136

Chapter 12 - Changing Data with the UPDATE Statement 138

Basic Syntax of UPDATE Statement..... 138
Updating a Single Column in a Single Row 140
The pitfall of using the UPDATE Statement 141
Changing Multiple Columns Using a Single UPDATE Statement 142
Updating Columns Based on Column Values from another Table..... 142
Using the .WRITE Clause with an UPDATE statement 143
Using the OUTPUT Clause with an UPDATE statement 146
Maintaining Data with the UPDATE Statements..... 148

Chapter 13 - How to Delete Rows from a Table 149

Creating Sample Data..... 149
Deleting all the Rows in a Table..... 150
Deleting Rows Based on a WHERE Condition..... 151
Deleting a Specific Number of Rows..... 153
Identifying Rows to Delete Based on another Table 153
Deleting Duplicate Rows using Common Table Expression..... 155
Using the Output Clause..... 157
Building Blocks for Deleting Rows..... 160

About the Author

Greg Larsen has worked in the computer industry since graduating from Washington State University in 1980, with a BS in Computer Science. Early on in his career Greg started managing databases which lead him to becoming a SQL Server DBA in 1999. Greg has written 100's of articles about SQL Server. Greg retired from fulltime DBA work in 2018. He now is an adjunct professor at St. Martins University, where he teaches SQL Server related subject matter. He has been active in the PASS community by running a user group and co-leading many SQL Saturday events. When not writing or being involved in the SQL community, Greg can be found on his sailboard somewhere along the western shores of Washington, British Columbia Canada and Alaska.

Introduction

Transact SQL (TSQL) code is used to maintain and unlock the knowledge of data stored in a SQL Server. The TSQL language is a proprietary version of the SQL language developed by Microsoft. TSQL is used to maintain, manipulate and report on data stored in SQL Server. This book will cover different aspects of the TSQL language. With an understanding of the basic TSQL language, programmers will have the building blocks necessary to quickly and easily build applications that use SQL Server.

This book takes the readers step by step through the TSQL language, one concept at a time. Each topic covers a particular aspect of TSQL with discussion and examples. The basics of the TSQL language are explored first, by starting with the SELECT statement. As the reader progresses through each chapter, they will learn about how to, join, sort, group, and aggregate data. In addition to the basic programming topics, this book also covers the history of SQL Server and the basics of set theory. By the end of the book, the readers will have a good foundation of how to write SELECT, UPDATE, INSERT and DELETE statements to manipulate SQL Server data.

Chapter 1

The Basic SELECT Statement

There are many different aspects of managing data in a SQL Server database. Before you can get into the complex issues associated with managing application data, you need to start with the basics of retrieving data from a table. To return data from a SQL Server table, you use a SELECT statement. In this chapter, I will cover the components of the basic SELECT statement and how you can use it to retrieve data from a single SQL Server table.

The Three Parts in a SELECT Statement

A basic SELECT statement that returns data from a single table consists of three different parts: The Column list, the FROM clause, and the WHERE Clause. The syntax for constructing a basic SELECT statement using these different components looks like Listing 1-1:

Listing 1-1: The SELECT statement syntax

```
SELECT <Column List>
FROM <table name>
WHERE <where criteria>;
```

The *<Column List>* will contain a list of columns that you want returned from the query, *<table_name>* will contain the table from which the data is selected, and the *<where criteria>* will identify the search criteria that will be used to constrain the rows that will be returned from the SELECT statement. Note that the WHERE clause is optional.

All of my examples in this Stairway will use the *AdventureWorks2017* SQL Server database that can be obtained from GitHub at this location: https://github.com/Microsoft/sql-server-samples/releases/tag/adventureworks.

Let's look at the following very simple SELECT statement in Listing 1-2 that selects some data from a table in the *AdventureWorks2017* database. You can follow along by running each of the query statement described in this article using the query window within SQL Server Management Studio while having the database set to *AdventureWorks2017*.

Listing 1-2: Your first SELECT statement

```
SELECT   ProductCategoryID ,
         Name
FROM     Production.ProductCategory
WHERE    ProductCategoryID < 2 ;
```

Here I selected two different columns, *ProductCategoryID* and Name, from the *Production.ProductCategory* table. Since this SELECT statement has a WHERE clause, it limits the rows returned from the table to only those rows that have a *ProductCategoryID* value less than 2.

Now that you have a basic idea of a SELECT statement, let me explore each of the components of the SELECT statement in a little more detail.

The Column List

The column list follows the SELECT keyword and is where you specify the columns you want to return from your table. The columns are identified by specifying the column name. If multiple columns are listed, they are separated by commas. In later chapters, we'll look at the possibilities for returning values other than columns from the specified table. In this chapter, we're sticking to the basics. Building on the example above, let's explore what a column list might look like if it only selects a single column or all of the columns of a table.

If I only wanted to return the *Name* column of the *Production.ProductCategory* table my query would look like Listing 1-3:

Listing 1-3: Returning just the NAME column

```
SELECT  Name
FROM    Production.ProductCategory
WHERE   ProductCategoryID < 2 ;
```

Here you can see I only specified only the *Name* column between the SELECT and FROM keywords in the above query. But if I wanted to specify all of the columns in the Production.ProductCategory table, I would run the following query from Listing 1-4:

Listing 1-4: Listing all columns

```
SELECT  ProductCategoryID ,
        Name ,
        rowguid ,
        ModifiedDate
FROM    Production.ProductCategory
WHERE   ProductCategoryID < 2 ;
```

In this SELECT statement, you can see that I have identified 4 different columns, each one separated by a comma. These columns can all be listed together on a single line or separated on different lines for readability, as I have done. Another way to include all the columns is to specify an asterisk instead of the individual column names. Listing 1-5 is a SELECT statement that uses the asterisk specification:

Listing 1-5: Using the asterisk to return all columns

```
SELECT  *
FROM    Production.ProductCategory
WHERE   ProductCategoryID < 2 ;
```

Care needs to be taken when using the asterisk (usually referred to as “star”) in your applications. Since the asterisk will return all columns in a table, if you happen to alter a table to include an additional column, then the additional column will be returned without modifying the actual SELECT statement. Whereas if you have specified all the columns by name, then when a new column is added, it will not be returned unless you add it to the column list. Using the asterisk is acceptable for testing, but if you want to follow best practices, do not use it in your application code. The reason this is a best practice is that most applications expect a fixed number of columns to be returned from a given SELECT statement. When columns have been added to a table, and you have used the asterisk method to identify table columns, the additional columns returned can break applications that have not been coded to handle this extra data.

The FROM Clause

In the FROM clause, you identify the table from which you want to select data. In this chapter, we are only discussing selecting data from a single table in the FROM clause. Note that as you progress in your knowledge of SQL, the FROM clause can identify multiple tables from which to select data.

There are several different ways to identify the table in which you want to select data. In fact, there are four different ways, three of which I will cover in this article and the fourth I will just mention.

The first way is to specify a table by identifying both the table name and the schema in which it belongs. This is how I have identified all my table names in all my examples so far. In each of my examples, I said the table name was *Production.ProductCategory*. The actual table name is just *ProductCategory,* and it is contained in the *Production* schema.

The second way to identify a table in the FROM clause is to just state the table name. When the FROM clause only contains the table name, SQL Server will assume the table is contained under the default schema for the database user, or under the dbo schema. Let me explain this concept in a little more detail.

When you submit a query to SQL Server that only identifies the table name, SQL Server will need to resolve which schema this table is under. This is because, in a given database, there can be multiple tables with the same name, as long as they are in different schemas. To determine which schema a table resides in SQL Server goes through a two-step process. The first step is to use the default schema for the database user who submitted the query and prepend their default schema the table name and then look for that table. If SQL Server finds the table using the user default schema, then step two is not performed. If SQL Server did not find the table using the user's default schema, then SQL Server checks in the *dbo* schema to find the table. Regardless of whether your database contains a single schema or not, it is best practice to get in the habit of identifying your tables in the FROM clause by using the schema name followed by the table name. By doing this, you reduce the amount of work SQL Server has to do to resolve the table name and promote plan cache re-usage.

The third way to identify a table is to use a three-part name, where the last two parts are schema and table name, and the first part is the database name. Here is a SELECT statement in Listing 1-6 similar to my previous SELECT statement that uses a three-part name:

Listing 1-6: Using a three-part name

```
SELECT  *
FROM    AdventureWorks2017.Production.ProductCategory
WHERE   ProductCategoryID < 2 ;
```

Here you can see that I prepend the database name *AdventureWorks2017* to the *Production.ProductCategory* table identified in the FROM clause.

By using a three-part naming convention for tables in the FROM clause, the database context of your query window in SQL Server Management Studio could be set to any database, and the database engine will still know which database, schema and table to use for the query. When building applications that need to retrieve data from multiple databases within an instance, using a three-part name helps facilitate retrieving data from multiple databases in a single application.

The last method is a using a four-part name, with the fourth part (preceding the database name) identifies the name of a linked server. Since linked servers are outside the scope of this Stairway, I will not discuss this topic further. If you should run across a table name that contains 4 parts, you will know the table is associated with a linked server.

The WHERE Clause

The WHERE clause of a SELECT statement is optional. The WHERE clause is used to constrain the rows that are returned from a SELECT statement. The database engine evaluates each row against the WHERE clause and then only returns rows if they meet the search condition or conditions identified in the WHERE clause. As you write more SELECT statements, you will find most of your SELECT statements will likely contain a WHERE clause.

A simple WHERE clause will contain a single search condition, whereas a more complex WHERE clauses might contain many conditions. When multiple conditions are used in a WHERE clause, they will be combined logically by using AND and OR logical operators. There is no limit to the number of different conditions that might be included in a SELECT statement. In my examples so far, I've only used a single condition. Let's review a couple of examples that have more complex WHERE clauses.

Listing 1-7 has a SELECT statement that has two different search conditions:

Listing 1-7: Two search conditions

```
SELECT  *
FROM    Production.Product
WHERE   Color = 'Blue'
        AND ProductID > 900 ;
```

In this statement, the first condition checks to see if the row has the value *Blue* in the *Color* column. The second condition checks to see if the value in the *ProductId* column is

greater than 900. Since the AND operator is used, both of these conditions must be true for a row to be returned from this query.

Let's look at a WHERE clause that gets even more complicated in Listing 1-8:

Listing 1-8: A complicated WHERE clause

```
SELECT  *
FROM    Production.Product
WHERE   ProductID > 900
        AND ( Color = 'Blue'
              OR Color = 'Green'
            ) ;
```

This example returns *Production.Product* rows where the *ProductID* value is greater than 900 and the value in the *Color* column is either *Blue* or *Green*. By looking at this query, you can see that I have placed parenthesis around the two different Color column conditions. This sets the order in which conditions are evaluated against each other, just like you would do in a mathematical expression. By using the parentheses, the OR operation between the two color conditions will be evaluated before evaluating the AND operator. When using AND and OR together without parentheses, the order in which these logical operators are processed is based on the rules of logical operator precedence. The precedence rules dictate that AND operations are performed before OR operations if no parentheses are included.

Let me build on my previous WHERE clause by adding a NOT operator in Listing 1-9:

Listing1- 9: Using NOT in the WHERE clause

```
SELECT  Name ,
        Color
FROM    Production.Product
WHERE   ProductID > 900
        AND NOT ( Color = 'Blue'
                  OR Color = 'Green'
                ) ;
```

I added the NOT operator just after the AND operation to indicate that I want products that are not Blue or Green. When I run this query against my *AdventureWorks2017* database, I get products that are Silver, Yellow and Black.

For a complete list of all the different possibilities of a search condition, refer to SQL Server Technical Documentation: https://docs.microsoft.com/en-us/sql/t-sql/queries/search-condition-transact-sql, and for more information on logical operator precedence you can read this topic: https://docs.microsoft.com/en-us/sql/t-sql/language-elements/operator-precedence-transact-sql?view=sql-server-ver15

Business Cases

Now it is your turn to use the information I've shared to build your own basic SELECT statements. In this section, I will be providing you with a couple of exercises with which you can practice writing a SELECT statement to meet the business case described.

Exercise #1:

Assume you have been asked by the head of the Human Resources department to produce a list of all *BusinessEntityID* values for employees who have lots of sick leave and vacation

hours. For this list, the Human Resources head tells you the definition of "lots of leave" is as follows: an employee must have a *SickLeaveHours* value greater than 68 and a *VacationHours* value greater than 98. You can find these columns along with the *BusinessEntityID* column in the *Human.Resources.Employee* table. Write and test your code against the *AdventureWorks2017* database. When done, check your code against the answer in the section below.

Exercise #2:

There are some questions your manager has about a few specific order numbers. Your manager wants you to produce a list of all the columns associated with a few specific orders so she can review which salesperson is associated with each order. Your manager specifies that she is only interested in a report that contains *SalesOrderHeader* rows for orders that have a *SalesOrderId* between 43702 and 43712. When you are done writing this SELECT statement, check your answer below.

Answers to Exercise #1:

Your query should look something like Listing 1-10:

Listing 1-10: Answer to Exercise #1

```
SELECT  BusinessEntityID ,
        SickLeaveHours ,
        VacationHours
FROM    HumanResources.Employee
WHERE   SickLeaveHours > 68
        AND VacationHours > 98 ;
```

The your query should return the following three rows:

```
EmployeeID  SickLeaveHours VacationHours
----------- -------------- -------------
109         69             99
179         69             99
224         69             99
```

Answers to Exercise #2:

Your query should look something like Listing 1-11 if you read the document referenced above to learn a little more about search conditions and the BETWEEN operator:

Listing 1-11: One answer to Exercise #2

```
SELECT  *
FROM    Sales.SalesOrderHeader
WHERE   SalesOrderID BETWEEN 43702 AND 43712 ;
```

Or you might have written your code similar to Listing 1-12:

Listing 1-12: An alternate answer to Exercise #2

```
SELECT  SalesOrderID ,
        RevisionNumber ,
        OrderDate ,
        DueDate ,
        ShipDate ,
        Status ,
        OnlineOrderFlag ,
        SalesOrderNumber ,
        PurchaseOrderNumber ,
        AccountNumber ,
        CustomerID ,
        SalesPersonID ,
        TerritoryID ,
        BillToAddressID ,
        ShipToAddressID ,
        ShipMethodID ,
        CreditCardID ,
        CreditCardApprovalCode ,
        CurrencyRateID ,
        SubTotal ,
        TaxAmt ,
        Freight ,
        TotalDue ,
        Comment ,
        rowguid ,
        ModifiedDate
FROM    Sales.SalesOrderHeader
WHERE   SalesOrderID >= 43702
        AND SalesOrderID <= 43712 ;
```

Your query should have returned 11 rows. Four rows have an *OrderDate* value of 2011-06-01, five rows have an *OrderDate* value of 2011-06-02, and two rows have an *OrderDate* value of 2011-06-03.

One thing to note about this solution versus the previous solution is that all the column names are identified. In contrast, the previous solution used the asterisk to specify all the columns. Identifying all the columns by name is a better coding practice because your SELECT statement will always return the same number of columns even if an additional column is added to your table. This is important when your application is expecting a specific number of columns to be returned from your SELECT statements.

Understanding the Basic SELECT Statement

In order to be a SQL Server application programmer, you need to understand the basic SELECT statement. Understanding how to retrieve all the data in a table and constraining what is returned is fundamental to developing applications. This article and exercises have provided you with the SELECT statement basics, which is the first building block of this series.

Chapter 2

History of Structured Query Language (SQL)

In Chapter 1, I gave you some information about how to write a basic SELECT statement. Now let's step back in time and discuss the history of *Structured Query Language*, or what most SQL Server professionals just shorten to SQL and pronounce like the second part of a bad movie and say *sequel*. Fasten your seatbelts while I crank up the time machine and travel back in time to follow the history of SQL and Microsoft SQL Server from its early years to where they are today.

The 1970s

It all began at a little known company called IBM in the early 1970s. A couple of researchers named Donald D. Chamberlin and Raymond F. Boyce developed the first incarnation of the SQL language while they were working in IBM's San Jose Research lab. They originally called this new coding language *SEQUEL*, which stood for ***S**tructured **E**nglish **Q**uery **L**anguage*. They invented this language to allowed programmers and infrequent database users to interact with data. The original SQL code set identified a set of functions and a set of simple and consistent rules. If you want to find out more information about the first *SEQUEL* rule set you can read a paper published by Chamberlin and Raymond, which can be found here: https://researcher.watson.ibm.com/researcher/files/us-dchamber/sequel-1974.pdf

It was discovered that the name SEQUEL was already trademarked by a United Kingdom aircraft company named Hawker Siddeley, which caused IBM to change the name of this new data manipulation language. They shortened the name to just three letters, SQL. That is how the SQL language was born.

In the late 1970s, a company called Relational Software Inc., which later became Oracle, saw the value of the relational database model and the SQL language developed by Chamberlin and Raymond. They started developing a database management system they hoped to sell to the U.S. Government. In June 1979 they released the first commercially available RDBMS that used SQL, which was called Oracle V2 which ran on a VAX machine.

Relational Software Inc. beat IBM to the marketplace with SQL. This wouldn't be the first or last time that a company beat IBM to the marketplace on a technology product. Shortly after the release of Oracle V2, IBM released its RDBMS known as System/38, which used the SQL language to manipulate data. System/38 proved to be a viable offering that led IBM to spend even more time and effort in exploring other software applications that took advantage of SQL.

The 1980s

In the 1980s, many other products that used SQL came to market. With many vendors exploiting SQL, the language was standardized by the American National Standards Institute (ANSI) who defined the SQL-86 standard in 1986. In 1987 the International Organization for Standards (ISO) adopted SQL. During this standardization process, the official pronunciation for SQL was declared to be *ess queue el*, but many people still refer to it as *sequel*.

With the SQL language now being a standard, vendor implementations of SQL found it hard to support their product lines with only standard SQL syntax. Therefore vendors started creating their own extensions to the SQL language to enhance their products. This is what led to Sybase to develop the Transact-SQL language extensions to support their own RDBMS implementation. Over time many of these original vendor-specific extensions would be adopted by other vendors, eventually finding their way into the standard SQL language.

In late 1987 Sybase and Microsoft launched into a partnership to produce and market DataServer, which used SQL and ran on the OS/2 operating system. At the time Ashton-Tate was the leader in PC databases with a product named dBase. In 1989, Microsoft went into a partnership with Ashton-Tate to release the first product that contained the name SQL Server,

with a product named Ashton-Tate/Microsoft SQL Server. A beta release of this product was shipped in the fall of 1988. This release was also called version 1.0 of SQL Server by some since the original name was a little too awkward for many people to refer to easily.

As with any standard, as it is used and expanded by different vendors and products, it needs to evolve. This was no different for SQL. In 1989 a new version of the ANSI/ISO SQL standard was established, which was dubbed SQL-89.

The 1990s

In the 1990s, vendors were providing products that used SQL. This continued support and expansion of the SQL language kept SQL moving forward in the software evolution process. The standards developed in the 80s were eventually reviewed, and a new standard emerged called SQL-92.

The SQL-92 standard expanded the standard SQL Language by leaps and bounds. SQL-92 brought in the concept of a schema manipulation language. This change introduced the ALTER and DROP commands into the SQL language. It also allowed for the dynamic creation of SQL statements and supported several new data types. Additional syntax was added to support outer joins as well as cascade updates and deletes. There were many other new features added to make the SQL language more flexible.

At the time SQL-92 was written, no one vendor adhered to the complete standard. Because of this, the standard was broken down into three different levels of compliance: entry, intermediate, and full. Each of these levels implemented a subset of the next higher level. The entry-level contained a subset of the functionality of the intermediate level, and the intermediate level contained a subset of the full standard. When buying products in the 1990s, vendors noted the level at which the product conformed to the standard.

In early 1990, Microsoft continued to enhance their 1988 release of SQL Server. In the summer of 1990, they release version 1.1. This version contained many bug fixes but also supported the just-released Windows 3.0 operating system. Microsoft continued to exploit

the SQL language when version 4.2 of SQL Server was released in 1993. This was the last release provided during the Sybase/Microsoft partnership.

In June of 1995, Microsoft released version 6.0 of SQL Server, which supported their new Windows 95 operating system. This release was then followed by SQL Server 6.5 in 1996. To round out the different versions released in the 90s Microsoft rolled out SQL Server 7.0 in 1998.

While Microsoft and other vendors continued rolling out new releases of their database software, the SQL standards continued evolving as well. In 1999 the standards organizations published SQL:1999. This latest standard included several new features to support the ever-changing SQL Server landscape. It was this version of the standard that incorporated support for large object types, user defined data types (UDT), established the SIMILAR and DISTINCT predicate, and many more new features. By the close of the century, SQL was becoming a commonly used language for storing and manipulating relational data.

The 2000s

The evolution of SQL didn't stop when the sands of time crossed over to the new millennium. Shortly after the beginning of the twenty-first century, Microsoft released the next version of SQL Server, codenamed Shiloh, or more commonly known as SQL Server 2000. As time marched on it took another 5 years before the next release of SQL Server, codenamed Yukon, which was released in 2005 with the name SQL Server 2005. Before the first decade of the new millennium was in the history books, Microsoft would release SQL Server 2008, staying with the year theme for their releases.

The 2010s

This new decade brought on big changes for SQL Server. With this decade, Microsoft introduced cloud based database solutions, and started rolling out new versions of SQL Server every 18 to 24 months. The first version to be released was 10.25. With this version of SQL Server, Microsoft started sharing the on-premise code base with a new service, known as

Azure SQL database. Azure SQL database ran in the cloud and was code named cloud DB. With the introduction of Azure SQL DB, a new paradigm for upgrade to SQL Server was developed. From this point forward all new enhancements would be rolled out in Azure SQL DB first, before they were incorporated into the on premise-version of SQL Server.

In 2010 version 10.5, code name Kilimanjaro, was also released under the name SQL Server 2008 R2. This was not an major upgrade to SQL Server but a minor one. This release was mainly a business intelligence release. SQL Server 2008 R2 introduced: Report Builder 3.0, Steaminsight, Master Data Services and more.

As promised, Microsoft kept up the excellerated rollout schedule of SQL Server by introducing Version 11, code name Denali in 2012. This release was known as SQL Server 2012. This version introduced column store index. This new index improve performance greatly for certain data warehouse queries.

The new on-premise versions of SQL Server kept coming. In 2014, SQL Server 2014, code name Hekaton was introduced. With this new release the Microsoft introduced the In-Memory OLTP engine. This allowed table to be stored in memory. The updatable column store index was also rolled out with this release.

Quicky behind SQL Server 2014 was the introduction of SQL Server 2016 (version 13). With this release Microsoft focused on rolling out some new security feature, such as Always Encrypted, and Dynamic Masking, It was also in this release that the query store was introduce, which is basically is a flight recorder for the difference query execution history.

Right behind SQL Server 2016 was the release of SQL Server 2017 (version 14), with a code name of Helsinki. With version 14, Microsoft expanded the supported operating systems for SQL Server to include Linux and Docker. Also the Adaptive Query Processing (ADP) feature was introduced. This new feature improved performance by introducing batch mode processing, memory grant feedback, interleaved executions, and the adaptive join operator.

The last release of this decade was SQL Server 2019 (version 15). With this release, the ADP was revamped and renamed to Intellegent Query Processing (IQP). The new features added to the IQP were, table variable deferred compilation, batch mode was expanded to support row storage structure, Scalar UDF inlining, and Rowmode memory grant feedback. Additional a new approximate count distinct fuction was added.

Just as Microsoft did in updating their older releases of SQL Server with a newer version, so did the governing body over the SQL language standards. During the first two decade of the new millennium four different versions of the SQL standards were published: SQL:2003,SQL:2008, SQL 2011, and SQL:2016. Each new release of SQL Standards brought new and enhanced features to the SQL language. These new standards provided many clarifications and minor modification to the already solid SQL language.

The Rest is History

The SQL language invented in the 70s was incorporated into the first version SQL Server in the late 80s. The rest of the journey from there to where we are at today is just history. SQL has come a long way since its early days at IBM. It is now widely used by many database vendors in one flavor or another. Microsoft SQL Server has also evolved into one of the leading contenders within the RDBMS space. The current millinum has already seen the excelleration of rolling out new releases of SQL Server. The future of SQL Server and the SQL language is not known, but I'm sure that SQL will continue to be refined and expanded.

To truly have an appreciation for the world of SQL, one needs to have at least a high-level understanding of relational database design. In my next chapter, I will be providing another history lesson about the father of relational database design (Edgar Frank "Ted" Codd), as well as a discussion of different database components related to database design.

Chapter 3

Implementing a Relational Model in SQL Server

In the previous chapters, I provided you with information about the basic SELECT statement and the history of SQL. Those chapters gave you a foundation for understanding how to retrieve data and how the SQL environment has evolved as technology and technical solutions have changed over time. In this chapter, I will be exploring how to implement a simple SQL Server database that is based off a relational model. Before jumping into creating a database, first, let me share a little history concerning the creator of the relational model.

The Father of Relational Data Modeling

The concept of relational database design was first introduced by Edgar F. Codd in 1970, with a paper titled “A Relational Model of Data for Large Shared Data Banks”. Codd developed this modeling theory while working at IBM. IBM didn’t jump fast enough on Codd’s data modeling concept and was, therefore, not the first vendor to supply a relational database engine that exploited Codd’s new relational data modeling theory. Codd’s relational modeling concept is now the framework that is used to create relational databases within SQL Server and other relational database engines.

Codd was born in the Isle of Portland in England and studied mathematics and chemistry before joining the Royal Air Force to become a World War II pilot. He moved to New York in 1948 and started working for IBM, where he was a mathematical programmer. He floated around for a number of years and then eventually moved to California to work at the IBM San Jose Research Laboratory. Codd continued working to refine and justify the relational data

model until the 1990s when his failing health forced him to retire. Edgar F. Codd died at the age of 79 on April 18, 2003.

Implementing a Relational Model in SQL Server

This book is not intended to teach you about relational data modeling, or database design. Instead, it is only to show you how to create a SQL Server database from a relational model. But before I can provide you with the code blocks for creating a SQL Server database, we first need to explore a relational data model that will be implemented. My simple model will contain a few entities (data tables) with primary key definitions and some relationships (foreign key constraints) between the different entities.

A primary key is a value that unique identifies a row in a table. The value of the key could be a single column value or could be made up of multiple column values. A foreign key is a column or multiple columns in a table that provide a link to another table. The foreign key columns in one table identify the primary key column in the other table.

My simple relational model will be for a simple hotel reservation system. This reservation system will need to track customer reservations information. Figure 3-1 illustrates this simple relational model that I will be implementing using T-SQL:

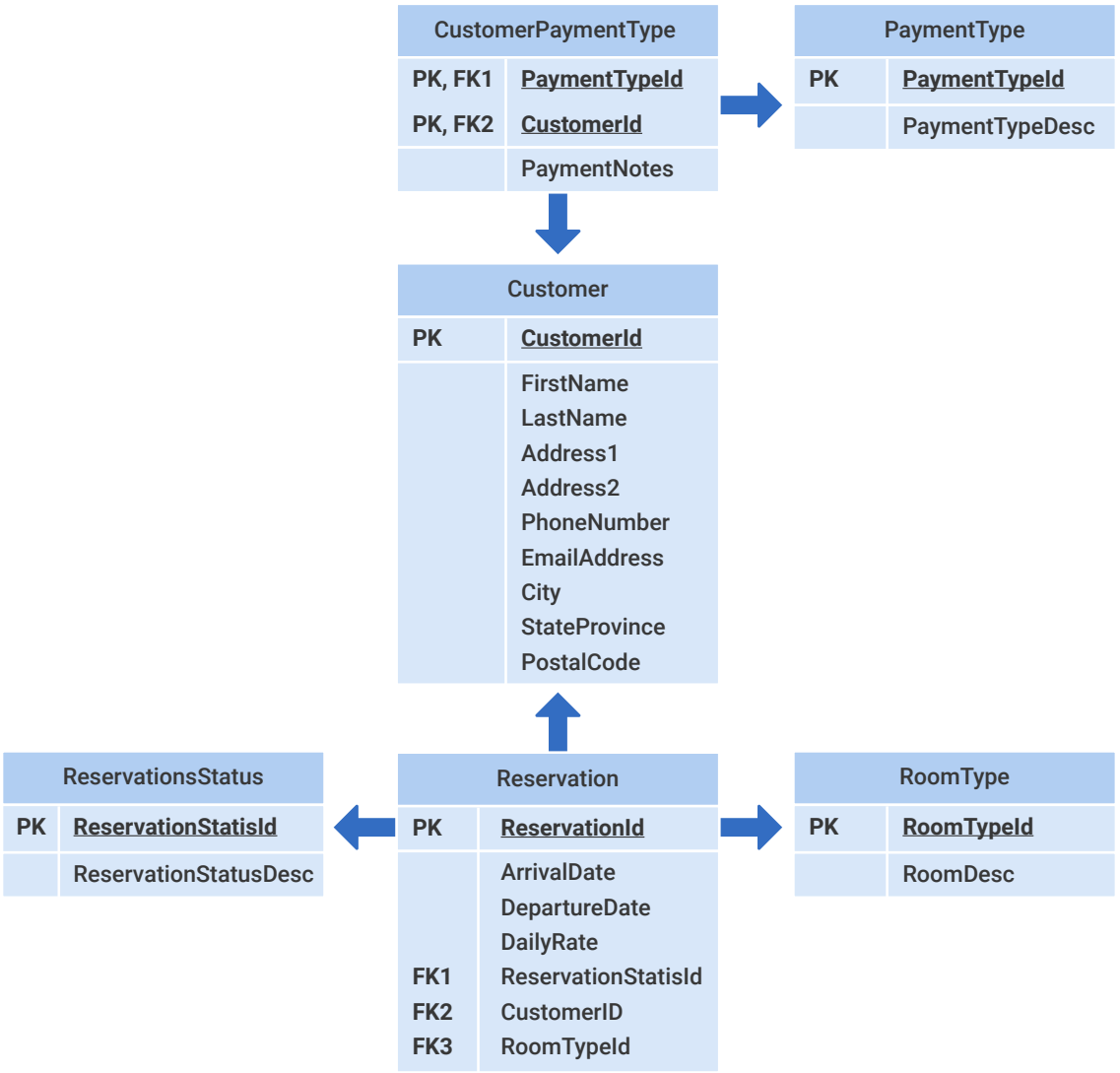


Figure 3-1: A simple relational database model consisting of 6 tables

By reviewing this model, you can see it contains several entities (represented by boxes) to track reservation-related information. Each entity is made up of a few attributes (columns) where one or more attributes are identified as the primary key (bold and underlined names). The arrows between the entities represent relationships between them. I will take this model

of entities, attributes, primary keys and relationships and develop a physical SQL Server database that represents the design of this relational model.

To build a physical database from this model, we need to identify the different objects in SQL Server that we are going to be defined based on this model. For each entity, or box in Figure 3-1, I will create a table in SQL Server. For each attribute of each entity, I will create a column in the associated table. For each primary key, I will create a unique clustered index (note a primary key could also be created using a unique non-clustered index. For more information on indexing refer to the Indexes Stairway at http://www.sqlservercentral.com/stairway/72399/). Lastly, for each relationship, I will create a foreign key constraint.

To get started, I first need to create a SQL Server database to hold all the new database objects I plan to create. My database will be called *RoomReservation*. I will create my database by using the following T-SQL code in Listing 3-1:

Listing 3-1: Code to create a database

```
CREATE DATABASE RoomReservation;
```

To start building *RoomReservation* database objects from my model, I will then create the table objects. To create a table in SQL Server, I need to use the `CREATE TABLE` statement. With the `CREATE TABLE` statement, I will be able to define each table and all of the columns in each table. Listing 3-2 has the simple syntax for creating a SQL Server table:

Listing 3-2: The CREATE TABLE syntax

```
CREATE TABLE <table_name> (
<column_definition> [,…N]);
```

Where:

<table_name> = Name of table

<column_definition> = column_name data_type,[NULL | NOT NULL]

For complete syntax of the `CREATE TABLE` statement refer to SQL Server documentation https://docs.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql?view=sql-server-ver15.

The first table I create will be the *Customer* table, created using the code in Listing 3-3.

Listing 3-3: Creating the Customer table

```
USE RoomReservation;
GO
CREATE TABLE dbo.Customer (
      CustomerId INT NOT NULL,
      FirstName NVARCHAR(50) NOT NULL,
      LastName NVARCHAR(50) NOT NULL,
      Address1 NVARCHAR(100) NOT NULL,
      Address2 NVARCHAR(100) NULL,
      PhoneNumber NVARCHAR(22) NOT NULL,
      EmailAddress VARCHAR(100) NULL,
      City VARCHAR(100) NOT NULL,
      StateProvince NVARCHAR(100) NOT NULL,
      PostalCode NVARCHAR(100) NOT NULL);
```

When I created my *Customer* table, I created all the columns I need, but I also specified whether or not the column must have a value when a record is inserted or updated into this table. I implemented that by specifying *NOT NULL* on some columns, while other columns I specified *NULL*.

Columns defined with NOT NULL do not allow you to create a record unless you populate the column with an actual value. Whereas defining a column using the *NULL* specification means you can create a row without specifying a value for this column. Another way to put it is that the column allows a NULL value. In my *CREATE TABLE* statement above, I allowed the columns *Address2* and *EmailAddress* to support NULLs, whereas all the rest of the columns require a value to be supplied when creating a row.

This `CREATE TABLE` statement did not completely define my *Customer* table as it is represented in my relational database model above. I still need to create a primary key constraint, which ensures that no two records in the table have the same *CustomerID*. The code to create that primary key is in Listing 3-4.

Listing 3-4: Adding a PRIMARY KEY constraint to the Customer table

```
USE RoomReservation;
GO
ALTER TABLE dbo.Customer ADD CONSTRAINT
     PK_Customer PRIMARY KEY CLUSTERED (CustomerId);
```

This `ALTER TABLE` statement added a primary key constraint to my *Customer* table. That primary key will be created in the form of a clustered index named *PK_Customer*.

In the Transact-SQL language, there is usually more than one way to do the same thing. Alternatively, I could have created my *Customer* table and primary key all at one time by running the *CREATE TABLE* statement in Listing 3-5.

Listing 3-5: An alternative method of creating the Customer table with PRIMARY KEY

```
USE RoomReservation;
GO
CREATE TABLE dbo.Customer (
      CustomerId INT NOT NULL CONSTRAINT PK_Customer PRIMARY KEY,
      FirstName NVARCHAR(50) NOT NULL,
      LastName NVARCHAR(50) NOT NULL,
      Address1 NVARCHAR(100) NOT NULL,
      Address2 NVARCHAR(100) NULL,
      PhoneNumber NVARCHAR(22) NOT NULL,
      EmailAddress NVARCHAR(100) NULL,
      City VARCHAR(100) NOT NULL,
      StateProvince NVARCHAR(100) NOT NULL,
      PostalCode NVARCHAR(100) NOT NULL);
```

At this point, I have shown you how to create a table with a defined PRIMARY KEY. The only thing left to show you is how to create a FOREIGN KEY constraint. But before I can do that, let me first provide you with the script to create the rest of the tables and PRIMARY KEYS in my relational database model above. You can find it in Listing 3-6.

Listing 3-6: Creating additional tables and PRIMARY KEY constraints

```
USE RoomReservation;
GO
CREATE TABLE dbo.Reservation (
      ReservationId INT NOT NULL,
      ArrivalDate DATETIME NOT NULL,
      DepartureDate DATETIME NOT NULL,
      DailyRate SMALLMONEY NOT NULL,
      ReservationStatusID INT NOT NULL,
      CustomerId INT NOT NULL,
```

```
      RoomTypeID INT NOT NULL);

ALTER TABLE dbo.Reservation ADD CONSTRAINT
    PK_Reservation PRIMARY KEY CLUSTERED (ReservationId);

CREATE TABLE dbo.RoomType (
      RoomTypeId INT NOT NULL,
      RoomDesc NVARCHAR(1000) NOT NULL);

ALTER TABLE dbo.RoomType ADD CONSTRAINT
    PK_RoomType PRIMARY KEY CLUSTERED (RoomTypeId);
CREATE TABLE dbo.ReservationStatus (
      ReservationStatusId INT NOT NULL,
      ReservationStatusDesc NVARCHAR(50) NOT NULL);
ALTER TABLE dbo.ReservationStatus ADD CONSTRAINT
    PK_ReservationStatus PRIMARY KEY CLUSTERED (ReservationStatusId);

CREATE TABLE dbo.PaymentType (
      PaymentTypeId INT NOT NULL,
      PaymentTypeDesc NVARCHAR(50) NOT NULL);
ALTER TABLE dbo.PaymentType ADD CONSTRAINT
    PK_PaymentType PRIMARY KEY CLUSTERED (PaymentTypeId);
CREATE TABLE dbo.CustomerPaymentType (
      PaymentTypeId INT NOT NULL,
      CustomerId INT NOT NULL,
      PaymentNotes NVARCHAR(2000) NULL);
ALTER TABLE dbo.CustomerPaymentType ADD CONSTRAINT
    PK_CustomerPaymentType PRIMARY KEY CLUSTERED (PaymentTypeId,
CustomerId);
```

A FOREIGN KEY constraint enforces referential integrity between two tables that are related to each other. The table that the foreign key constraint is defined on is the 'referencing table'. It is required to have a related record in another table, known as the 'referenced' table, any time a row is inserted or updated in the table. In my relational model in Figure 3-1, these foreign key relationships are represented by the arrows. FOREIGN KEY constraints are only defined on one of the tables in the relationship. In my diagram, the FOREIGN KEY constraints will be defined on those tables that have the tail of the arrow (non-pointed end) attached.

To define these FOREIGN KEY constraints in my relational model, I will need to alter each referencing table to add the constraint. Listing 3-7 is the T-SQL code I can use to create a FOREIGN KEY constraint on the *Reservation* table. This constraint ensures that a record does not get inserted or updated in the *Reservation* table unless a matching record is found in the *Customer* table, based *on CustomerId*.

Listing 3-7: Creating a FOREIGN KEY constraint on the Reservation table referencing the Customer table

```
USE RoomReservation;
GO
ALTER TABLE dbo.Reservation
ADD CONSTRAINT FK_Reservation_CustomerPaymentType FOREIGN KEY
(CustomerId)
    REFERENCES dbo.Customer (CustomerID);
```

To complete my design, I need to implement all the other foreign key constraints identified in my model in Figure 3-1. Listing 3-8 contains the `ALTER TABLE` statements to create the additional foreign key constraints in my data model.

Listing 3-8: Creating additional FOREIGN KEY constraints

```
USE RoomReservation;
GO
ALTER TABLE dbo.Reservation
ADD CONSTRAINT FK_Reservation_RoomType FOREIGN KEY (RoomTypeId)
    REFERENCES dbo.RoomType (RoomTypeId);

ALTER TABLE dbo.Reservation
ADD CONSTRAINT FK_Reservation_ReservationStatus FOREIGN KEY
(ReservationStatusId)
    REFERENCES dbo.ReservationStatus (ReservationStatusId);

ALTER TABLE dbo.CustomerPaymentType
ADD CONSTRAINT FK_CustomerPaymentType_PaymentType FOREIGN KEY
(PaymentTypeId)
    REFERENCES dbo.PaymentType (PaymentTypeId);

ALTER TABLE dbo.CustomerPaymentType
ADD CONSTRAINT FK_CustomerPaymentType_Customer FOREIGN KEY (CustomerId)
    REFERENCES dbo.Customer (CustomerId);
```

Validating Database Design

Once I am are done building a database from a data model, I should validate the implemented design to make sure it is correct. This validation process is to ensure all the data integrity rules I built into my physical database are implemented correctly. In my design here are the rules I need to validate

- All rows inserted or updated must have a specific value defined for any column defined as `NOT NULL`.
- Columns that are *PRIMARY KEY*s do not allow duplicate values
- Columns that have foreign key constraints do not allow data that does not have a matching record in the referenced table

Before I can validate the data integrity rules, I first need to populate the referenced tables with some valid data. I will use the code in Listing 3-9 to populate those tables with some valid data:

Listing 3-9: Inserting initial data

```
USE RoomReservation;
GO
SET NOCOUNT ON;
-- Create PaymentType records
INSERT INTO PaymentType VALUES(1,'Visa');
INSERT INTO PaymentType VALUES(2,'MasterCard');
INSERT INTO PaymentType VALUES(3,'American Express');
-- Create Customer
INSERT INTO Customer VALUES
    (1,'Greg','Larsen','123 Some Place'
      ,NULL,'123-456-7890',Null,'MyCity','MA','12345');
-- Create Reservation Status
INSERT INTO ReservationStatus VALUES (1,'Booked');
INSERT INTO ReservationStatus VALUES (2,'Cancelled');
-- Create Room Type
INSERT INTO RoomType VALUES (1,'Kingsize');
INSERT INTO RoomType VALUES (2,'Queen');
INSERT INTO RoomType VALUES (3,'Double');
```

To validate the data integrity rules that I built into my database, I will be running the *INSERT* statements in Listing 3-10.

Listing 3-10: Testing various constraints with INSERT statements

```
USE RoomReservation;
GO
-- Violates NOT NULL constraint
INSERT INTO Reservation VALUES(1,'2011-8-1 5:00 PM'
                               ,'2011-8-2 9:00 AM'
                               , 150.99,NULL,1,1);
-- Violates Primary Key Constraint
INSERT INTO RoomType VALUES (3,'Suite');
-- Violates Foreign Key Constraint
INSERT INTO CustomerPaymentType VALUES (1,2,'Will need an internet
connection');
```

Each one of these `INSERT` statements should fail because they violate a data integrity rule that was built into the *RoomReservation* database. The first `INSERT` statement violates the `NOT NULL` validation check of the *ReservationStatusID* column.

The second `INSERT` statement violates the PRIMARY KEY constraint that was placed on the *RoomType* table. This `INSERT` statement is trying to insert the value of *3* for the *RoomTypeID* column. The problem is there is already a record in the *RoomType* table with *RoomTypeID* value of 3.

The last `INSERT` statement violates the FOREIGN KEY constraint of the *CustomerPaymentType* table. In this particular `INSERT` statement, there is no *CustomerID* with a value of *2* in the *Customer* table.

To correctly insert these records, the inserted data values will need to be cleaned up. Once the data has been cleaned up, I will be able to insert this new data into the appropriate tables.

Listing 3-11 contains the cleaned-up `INSERT` statements that will pass all data integrity checks and be successfully inserted into the appropriate tables in the *RoomReservation* database:

Listing 3-11: Additional constraint testing

```
USE RoomReservation;
GO
-- Violates NOT NULL constraint
INSERT INTO Reservation VALUES(1,'2011-8-1 5:00 PM'
                               ,'2011-8-2 9:00 AM'
                               , 150.99,1,1,1);
-- Violates Primary Key Constraint
INSERT INTO RoomType VALUES (4,'Suite');
-- Violates Foreign Key Constraint
INSERT INTO CustomerPaymentType VALUES (1,1,'Will need an internet
connection');
```

Relational Database Design

My reservation example demonstrates how to take a relational model and use it to implement a SQL Server database. By using the NOT NULL, PRIMARY KEY, and FOREIGN KEY constraints, I built data integrity rules right into my database design. This allowed me to enforce these rules in the underlying database definition, instead of having to write code in my business processing layer to validate these data rules. By doing this, I allowed the SQL Server database engine to perform these data integrity checks for me.

By understanding and creating your database design around the relational database model, you will be building a robust and efficient database implementation, where you can build data integrity checking right into the database.

Chapter 4

The Mathematics of SQL: Part 1

A relational database contains tables that relate to each other by key values. When querying data from these related tables, you may choose to select data from a single table or many tables. If you select data from many tables, you usually join those tables together using specified join criteria. The concepts of selecting data from tables and joining tables together are all about managing and manipulating sets of data. In this chapter, I will explore the concepts of set theory and mathematical operators to join, merge, and return data from multiple SQL Server tables.

What is a table?

What is a table? A table is just a bunch of rows that contain values in one or more columns. A table can be thought of as a set containing zero, one, or many records. Each row in a table is considered a member of the table set. The concept of sets and members of sets are the basic elements of set theory. By performing set operations, such as intersection and union against different sets (or against different tables in a SQL Server database), we can retrieve the data we need to support our applications.

Basic Set Theory

In the late 1800's Georg Cantor formally defined a set. He defined a set as "any collection into a whole **M** of definite and separate objects **m**.", where the separate objects **m** are elements of the set **M**. Cantor's work eventually grew into what we now call set theory. Edgar Codd used the concept of set theory to explain the relationships between data in database tables, which formed the basis of our all familiar relational database. The relational database concept that Edgar Codd developed was just a projection of the mathematical concepts of set theory for

retrieving data stored in a database. By better understanding set theory, we can better understand and manipulate our data that is stored in our relational databases.

In addition to a set, there is also the concept of a subset. A subset is just a collection of members of a set where the number of members in the subset is less than the number members of the whole set. A subset can be thought of as a partial list of members of a set. In SQL Server, a subset of a table would be a set of rows, where the total number of rows in the subset has fewer rows than the original table.

To retrieve data for your application, you are typically doing more than just retrieving a set of records from a single SQL Server table. You will commonly need to join one set with another set, and then finally use some parts of that joined set to support the data that your application might need. To explore the idea of joining sets together a little more, I will explain a concept known as Venn diagrams.

Venn Diagrams

A Venn diagram is a graphical representation of how two or more sets intersect each other. A Venn diagram makes it easy to visualize and describe the different parts of intersecting sets. The Venn diagramming concept was invented by John Venn, although he never called it a Venn diagram. Figure 4-1 shows a simple Venn diagram that represents the intersection of two sets, ‘Set A’ and ‘Set B’.

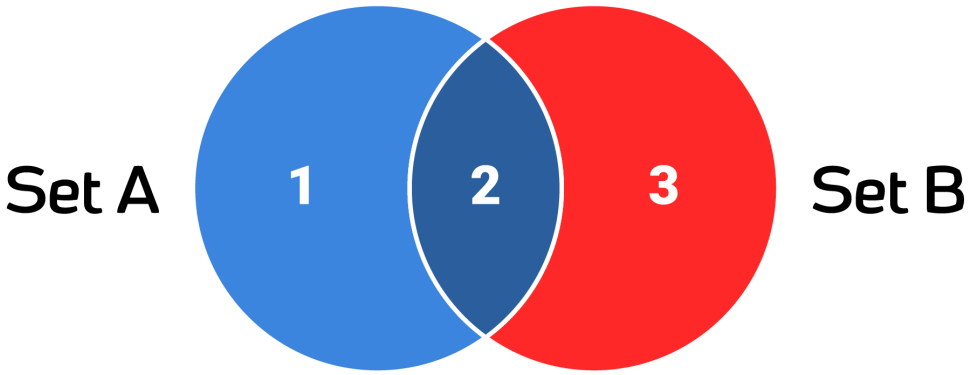


Figure 4-1: Simple Venn diagram that represents two intersecting sets

Set A is represented by the light blue circle, and *Set B* is represented by the red circle. I have labeled the different parts of the joined set with 1, 2, and 3, as well as shaded each part with a different color so that I can discuss each of these parts. The light blue part of Figure 4-1 that is labeled '1' represents the members in *Set A* that do not match or intersect with any members in *Set B*. The purple part, labeled '2', contains the members of *Set A* that intersect, or have the matching members in *Set B*. The red part labeled '3' is the part of *Set B* that has no matching, or intersecting members with *Set A*.

A Venn diagram can be used to represent how different sets intersect visually. Figure 4-2 shows a Venn diagram that represents where *Set B* is contained within *Set A*, so B is a subset of A.

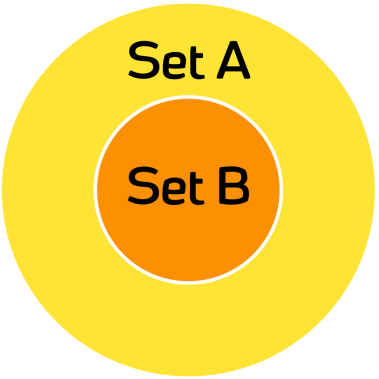


Figure 4-2: Venn diagram representing a subset

Venn diagrams can even be more complex; they can represent the intersection of multiple sets. Figure 4-3 shows the intersection of three different sets.

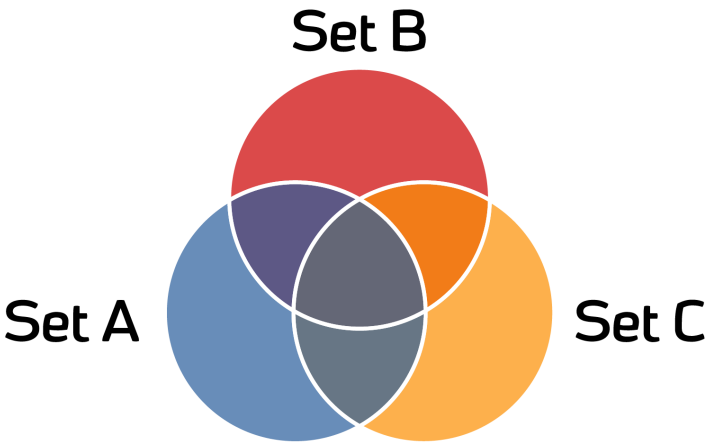


Figure 4-3: Venn diagram showing the Intersection of 3 Sets

Understanding of the different parts of intersecting sets as represented using Venn diagrams can give you a better sense for how joined sets can be used to support retrieving data from relational databases.

Mathematical Operators

You might remember *INTERSECT* and *UNION* operators from your mathematical studies of algebra. These mathematical operators have been around since set theory was introduced, and they not only apply to joining sets, but they also apply to joining SQL Server tables. When we talk about joining tables in SQL Server, we think of using INNER JOIN and OUTER JOIN operators. These operators will be discussed in the next chapter.

There is also another operator that is available in SQL Server that I will be discussing that was not part of your algebraic mathematical studies, and that is the *EXCEPT* operator. This operator is useful for returning part of an intersecting set and therefore is worthy of discussion along with *INTERSECT* and *UNION*.

To help demonstrate how each of these operators works, I will walk you through various *SELECT* statements that use the *INTERSECT*, *UNION* and *EXCEPT* operators. I will also use Venn diagrams and a few simple SQL Server tables to explore how these operators can be used to return a set of rows when joining SQL Server tables.

For the first few examples, I will be using two different sets: *Set A* and *Set B*. These two sets and their members are shown in Figure 4-4.

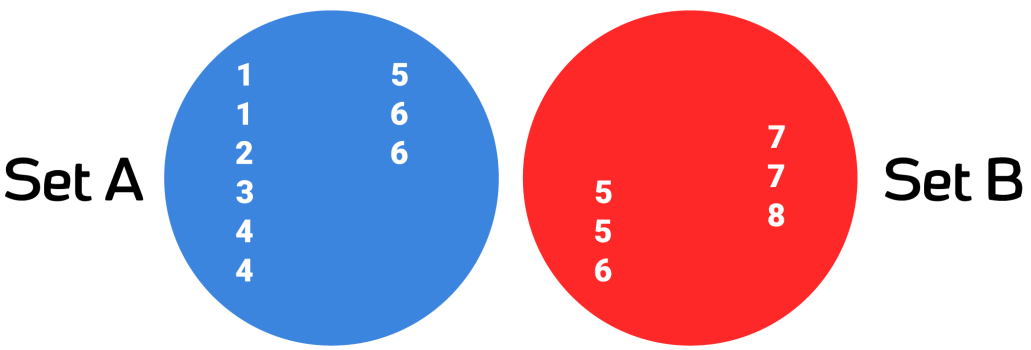


Figure 4-4: Set A and Set B

In Figure 4-4, *Set A* contains 9 members consisting of numbers ranging in value from 1 to 6. Some members are unique, and others are not. *Set B* has 6 members, ranging in value from 5 to 8. Some of the members are unique and some that are not. The Venn diagram in Figure 4-5 shows the intersection of these two sets based on those members in *Set* A that have matching values to the members in *set B*.

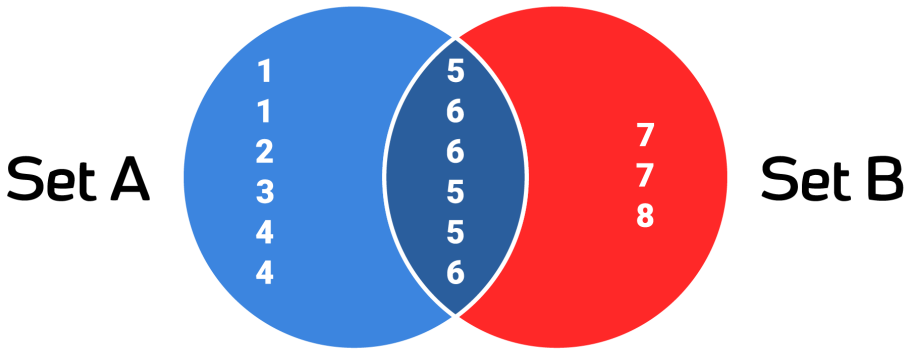


Figure 4-5: [Set A] and [Set B] intersect on members that have matching values.

In Figure 4-5, you can visually see the different parts of the two intersecting sets. Part 1 consists of members with values of 1 through 4, part 2 members have values of 5 and 6, and part 3 contains members with the values of 7 and 8. I will demonstrate how the *INTERSECT*, *UNION* and *EXCEPT* operators can be used to return these different parts using rows in SQL Server tables. To do this, I will first build SQL Server tables that contain the two sets and their members identified in Figure 4-4. The code is shown in Listing 4-1.

Listing 4-1: Creating sample data for Set A and Set B

```
USE tempdb;
GO
SET NOCOUNT ON;
-- Create Set A
CREATE TABLE [SET A] (Member TINYINT);
INSERT INTO [SET A] VALUES (1);
INSERT INTO [SET A] VALUES (1);
INSERT INTO [SET A] VALUES (2);
INSERT INTO [SET A] VALUES (3);
INSERT INTO [SET A] VALUES (4);
INSERT INTO [SET A] VALUES (4);
INSERT INTO [SET A] VALUES (5);
INSERT INTO [SET A] VALUES (6);
INSERT INTO [SET A] VALUES (6);
-- Create Set B
CREATE TABLE [SET B] (Member TINYINT);
INSERT INTO [SET B] VALUES (5);
INSERT INTO [SET B] VALUES (5);
INSERT INTO [SET B] VALUES (6);
INSERT INTO [SET B] VALUES (7);
INSERT INTO [SET B] VALUES (7);
INSERT INTO [SET B] VALUES (8);
```

The *INTERSECT* operator combines two sets based on the common members between the two sets, just like the Venn diagram did in Figure 4-5. The result of the *INTERSECT* operation is a list of the common members between the two intersecting sets. In my example, I will use the *INTERSECT* operator to return the members in part 2 of the Venn diagram from Figure 4-5. The difference is that the members will be distinct (duplicates will be removed), so the result set only returns a single row for every unique member value. Removing the duplicates is something that the *INTERSECT* operator does automatically. To demonstrate how the *INTERSECT* operator works, I run will execute the code in Listing 4-2.

Listing 4-2: Intersection of Set A and Set B

```
USE tempdb;
GO
SELECT Member FROM [SET A]
INTERSECT
SELECT Member FROM [SET B];
```

The results of running Listing 4-2 can be found in Report 4-1. By reviewing Report 4-1, you can see how only two rows are returned, and the values returned are those distinct values that are common between tables *[SET A]* and *[SET B]*.

Report 4-1: Output of intersection operation between [SET A] and [SET B]

```
Member
------
5
6
```

The *UNION* operator brings together all three different parts (1, 2, and 3) from the Venn diagram in Figure 4-5. It returns every member value in both *[SET A]* and *[SET B]* and de-duplicates the members while the union operation is performed. To demonstrate how the *UNION* operator works, I will run the code in Listing 4-3.

Listing 4-3: Code to join two sets together using the UNION operator

```
SELECT Member FROM [SET A]
UNION
SELECT Member FROM [SET B];
```

When this code is run, the results in Report 4-2 are produced. If you look at the results in this report, you can see all the member values are returned, but all the duplicate member values in the two different sets have been removed.

Report 4-2: Results from UNION operation on [SET A] and [SET B]

```
Member
------
1
2
3
4
5
6
7
8
```

It is possible to retain the duplicate values while the union operation is performed against two sets. This can be done by using the *UNION ALL* operator. The '*ALL*' part of the operator tell SQL Server not to remove the duplicates. To demonstrate the *UNION ALL* operator, I will run the code in Listing 4-4.

Listing 4-4: Example of using UNION ALL operator

```
SELECT Member FROM [SET A]
UNION ALL
SELECT Member FROM [SET B];
```

When I run the code in Listing 4-4, the results in Report 4-3 are produced. You can see that all the members of both tables [SET A] and [SET B] are returned, including the duplicates.

Report 4-3: Output from using the UNION ALL operator

```
Member
------
1
1
```

```
2
3
4
4
5
6
6
5
5
6
7
7
8
```

All of my examples so far have shown INTERSECT and UNION operations being performed against only two sets. It is possible to use *INTERSECT* or *UNION* to combine more than just two sets. To demonstrate this, let me first add another table called [SET C] by running the code in Listing 4-5.

Listing 4-5: Adding table [SET C]

```
CREATE TABLE [SET C] (Member TINYINT);
INSERT INTO [SET C] VALUES (5);
INSERT INTO [SET C] VALUES (5);
INSERT INTO [SET C] VALUES (9);
INSERT INTO [SET C] VALUES (8);
```

By using the tables *[SET A]*, *[SET B]* and *[SET C]* and joining these tables together using the INTERSECT operator, I am producing the result set represented by the center section in the Venn diagram shown in Figure 4-3. To demonstrate intersecting my three different sets, I will run the code in Listing 4-6.

Listing 4-6: Intersecting 3 different sets

```
SELECT Member FROM [SET A]
INTERSECT
SELECT Member FROM [SET B]
INTERSECT
SELECT Member FROM [SET C];
```

The results of running the code in Listing 4-6 can be found in Report 4-4. You can see that only one member with a value of 5 is returned. The member value of 5 is the only member that is common across all three sets.

Report 4-4: Results of intersecting [SET A], [SET B], and [SET C]

```
Member
------
5
```

If you are looking to test what you have learned so far, you can try to build the code to *UNION* together *[SET A], [SET B]* and *[SET C]*. This will be relatively easy to do, by just adding another *UNION* operator and a *SELECT* statement to the code in Listing 4-3.

EXCEPT operator

The *EXCEPT* operator is quite different than either *INTERSECT* or *UNION*. It brings back only the left portion of two joined sets or part 1 of the Venn diagram in Figure 4-1. However, there is one similarity between the *EXCEPT* operator and the *INTERSECT* and *UNION* operators. The *EXCEPT* operator de-duplicates the members in its result set. To demonstrate this, I can run the code in Listing 4-7.

Listing 4-7: De-duplicating Part 1 members using the EXCEPT operator

```
SELECT Member FROM [SET A]
EXCEPT
SELECT Member FROM [SET B];
```

When I run the code in Listing 4-7, I get the results in Report 4-5. The results in the report only show the unique member values from [SET A] that are not in [SET B] or the members in part 1 of the Venn diagram in Figure 4-5.

Report 4-5: Results of joining two sets using the EXCEPT operator

```
Member
------
1
2
3
4
```

If you want to bring back part 3 of the Venn diagram in Figure 4-5, you can do that, but the *SELECT* from table [SET B] needs to be listed first, and the *SELECT* from table [SET A] needs to be listed second, with the *EXCEPT* operator between the two *SELECTs*. The order of the *SELECTs* needs to be reversed since the *EXCEPT* operator only brings back members from the first *SELECT*, that are not contained in the second *SELECT*. I will leave it up to you to build and execute the code to return part 3 of the Venn diagram in Figure 4-5.

Joining Tables that Contain More than a Single Column

The *INTERSECT* and *UNION* statements can be used to join tables that have more than a single column, as my previous examples have shown. To make this work each SELECT list needs to contains the same number of columns, and those columns from each table need to be compatible data types. To demonstrate this, I will first create two tables (*[SET D]* and *[SET E]*) that contain multiple columns by using the code found in Listing 4-8.

Listing 4-8: Code to create two tables with multiple columns

```
SET NOCOUNT ON;
-- Create Set D
CREATE TABLE [SET D] (Col1 TINYINT,
                      Col2 CHAR(1));
INSERT INTO [SET D] VALUES (1,'A');
INSERT INTO [SET D] VALUES (1,'B');
INSERT INTO [SET D] VALUES (2,'A');
INSERT INTO [SET D] VALUES (3,'A');
INSERT INTO [SET D] VALUES (4,'A');
INSERT INTO [SET D] VALUES (4,'A');
INSERT INTO [SET D] VALUES (5,'A');
INSERT INTO [SET D] VALUES (6,'A');
INSERT INTO [SET D] VALUES (6,'A');
-- Create Set E
CREATE TABLE [SET E] (Col1 TINYINT,
                      Col2 CHAR(1),
                      Col3 CHAR(1));
INSERT INTO [SET E] VALUES (5,'Z','A');
INSERT INTO [SET E] VALUES (5,'Y','C');
INSERT INTO [SET E] VALUES (6,'X','B');
INSERT INTO [SET E] VALUES (7,'W','A');
INSERT INTO [SET E] VALUES (7,'V','A');
INSERT INTO [SET E] VALUES (8,'U','A');
```

After creating these two tables, I will then use the code in Listing 4-9 to intersect tables *[SET D]* and *[SET E]*. Note that I have not used 'Col2' of *[SET E]* but instead used 'Col3'. I did this to demonstrate that the *INTERSECT* operation will still work, even without both *SELECTs* returning the same column names. The *UNION* and *INTERSECT* will work as long as both *SELECTs* used return the same number of columns.

Listing 4-9: Intersecting tables that contain multiple columns

```
SELECT Col1, Col2 FROM [SET D]
INTERSECT
SELECT Col1, Col3 FROM [SET E];
```

When I run the code in Listing 4-9, I get the results found in Report 4-6.

Report 4-6: Results of intersecting two tables with multiple columns

```
Col1 Col2
---- ----
5    A
```

The de-duplication process removes duplicates by looking at all the columns returned for a row. That is, two result rows are only considered to be duplicates if all the columns in each of the rows have matching values. By reviewing the output in Report 4-6, you can see that only the unique rows between *[SET D]* and *[SET E]* were returned when I ran the code in Listing 4-9. In addition, something worth noting is that the column headers displayed take the names associated with the columns identified in the first SELECT.

Set Theory and SQL Server

Set theory, which has been around since the late 1800s, laid the groundwork for selecting information out of SQL Server databases. With the use of Venn diagrams, we can show visually how two or more sets intersect. Knowing how our 19th-century mathematicians defined and joined sets is useful in helping us understand how to retrieve data from SQL Server tables using the *INTERSECT, UNION, UNION ALL* and *EXCEPT* operators.

Chapter 5 will be Part 2 of "The Mathematics of SQL." In Part 2, I will explore the INNER JOIN and OUTER JOIN operators and how these operators can be used to bring back different parts of a Venn diagram as a result set.

Chapter 5

The Mathematics of SQL: Part 2

In the last chapter, I introduced you to the concept of Venn diagrams and discussed using the *INTERSECT*, *UNION* and *EXCEPT* operators. This stairway will continue exploring the mathematics of SQL by discussing the two different join operators, *INNER JOIN* and *OUTER JOIN*.

Joining tables is crucial to understanding data relationships in a relational database. When you are working with your SQL Server data, you will often need to join tables to produce the results your application requires. Having a good understanding of set theory, the mathematical operators available, and how they are used to join tables will make it easier for you to retrieve the data you need from SQL Server.

Joining Two Sets

A set is a well-defined, unordered list of members. When two sets are joined together, the results of the joined sets contains three different parts, as represented by the Venn diagram in Figure 5-1. The *INNER JOIN* and *OUTER JOIN* operations can be used to return the different parts of the Venn diagram in Figure 5-1. The *INNER JOIN* operator can be used to return the members (or rows) represented by part 2 of the Venn diagram in Figure 5-1. The *OUTER JOIN* operator can be used to return the members from the other two parts (part 1 and 3) in Figure 5-1.

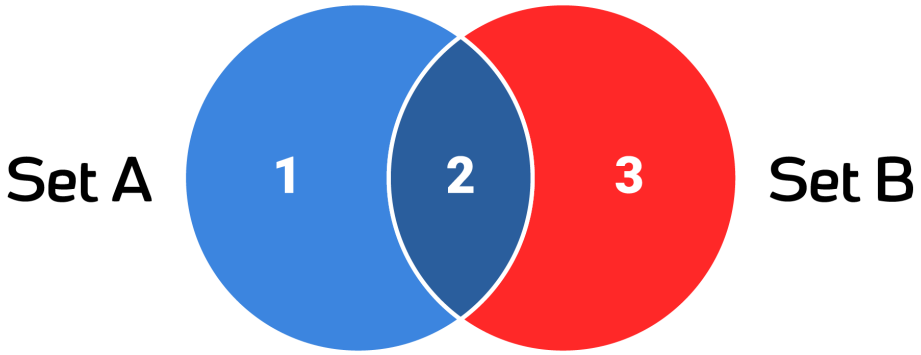


Figure 5-1: Different Parts of two-joined sets

To show what is returned when using these two different join operators, I am going to use two sets (implemented as SQL Server tables) that are represented in Figure 5-2.

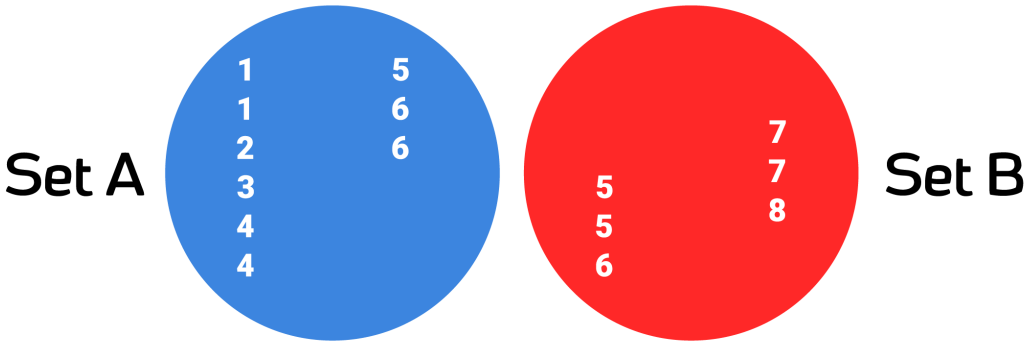


Figure 5-2: Sample data for [SET A] and Set B

In Figure 5-2, *[Set A]* contains 9 members represented by the numbers ranging from 1 to 6, where some members are unique, and others are not. *[Set B]* has 6 members ranging from 5 to 8 and has some unique members and some that are not as well. I am going join these two sets on the members in *[Set A]* that are equal to the members in *[Set B]*, which is represented by the Venn diagram in Figure 5-3.

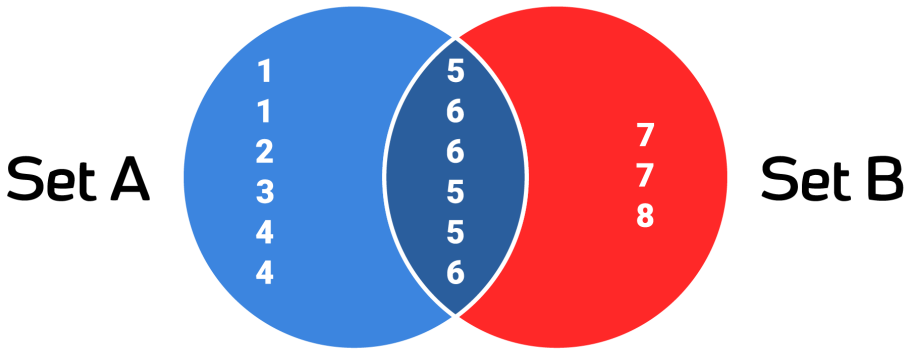


Figure3: [Set A] and [Set B] joined on members that are equal.

By looking at the Venn diagram of the joined sets in Figure 5-3, you can visually see that it is joined based on the member values that are same between *[SET A]* and *[SET B].* This is represented by the center section or part 2 of the Venn diagram. The members in *[SET A]* that are not included in *[SET B]* have member values 1 through 4 and are represented by the left part of the Venn diagram, or what I am calling part 1. The members in *[SET B]* that are not in *[SET A]* are those members in the right (or red) section of the Venn diagram in Figure 5-3. These members are the values 7 and 8. I will show you how to use the *INNER JOIN* and *OUTER JOIN* operators to return the members in each of these different parts.

Creating Sample Data for [SET A] and Set B

To demonstrate how to return these different parts of joined sets using the join operators, I will build a couple of SQL Server tables that contain the members of the different sets described in Figure 5-2 and Figure 5-3.To do that I will use the code in Listing 5-1.

Listing 5-1: Creating sample data for [SET A] and Set B

```
USE tempdb;
GO
SET NOCOUNT ON;
-- Create Set A
DROP TABLE IF EXISTS [SET A];
CREATE TABLE [SET A] (Member TINYINT);
INSERT INTO [SET A] VALUES (1);
INSERT INTO [SET A] VALUES (1);
INSERT INTO [SET A] VALUES (2);
INSERT INTO [SET A] VALUES (3);
INSERT INTO [SET A] VALUES (4);
INSERT INTO [SET A] VALUES (4);
INSERT INTO [SET A] VALUES (5);
INSERT INTO [SET A] VALUES (6);
INSERT INTO [SET A] VALUES (6);
-- Create Set B
DROP TABLE IF EXISTS [SET B];
CREATE TABLE [SET B] (Member TINYINT);
INSERT INTO [SET B] VALUES (5);
INSERT INTO [SET B] VALUES (5);
INSERT INTO [SET B] VALUES (6);
INSERT INTO [SET B] VALUES (7);
INSERT INTO [SET B] VALUES (7);
INSERT INTO [SET B] VALUES (8);
```

INNER JOIN Operator

The *INNER JOIN* operator is used to return part 2 of the Venn diagram in Figure 5-3. By using the *INNER JOIN* operator to join *[SET A]* and *[SET B]* , I can return the member values of 5 and 6. Listing 5-2 contains the T-SQL code to perform this *INNER JOIN* operation.

Listing 5-2: Demonstration of INNER JOIN

```
-- INNER JOIN
SELECT [SET A].Member AS "[SET A] Member"
      ,[SET B].Member AS "[SET B] Member"
FROM [SET A] INNER JOIN [SET B]
ON [SET A].Member = [SET B].Member;
```

When the code in Listing 5-2 is run the output in Report 5-1 is returned.

Report 5-1: Output when running code in Listing 5-2

```
[SET A] Member [SET B] Member
-------------- --------------
5              5
5              5
6              6
6              6
```

The *INNER JOIN* operator returns the member values of 5 and 6 and doesn't remove duplicates. This *INNER JOIN* operation does not return 6 different rows, as shown in the Venn diagram in Figure 5-3. The reason for this is that the Venn diagram is only the logical representation of the *INNER JOIN* operation. In contrast, the output in Report 5-1 is the result of the actual *INNER JOIN* operation. When the SQL Server engine performs an INNER JOIN operation, it takes each member of *[SET A]* and matches it with members from [SET B] that have the same value. The only members that have matching values between *[Set A]* and *[SET*

B] are the members with a value of 5 or 6. The INNER JOIN operation works like this; it first takes the single member with a value of 5 from *[SET A]* and matches members in *[SET B]* that also have a value of 5. For each match between two sets, one row is returned. This yields the two rows that have member values of 5 in Report 5-1. The engine also performs the same INNER JOIN operations for each member in *[SET A]* that has a matching member in *[SET B]* with a value of 6. Since there are two members in *[SET A]* with a value of 6 and only one member in *[SET B]* with a value of 6 only two rows are returned and displayed in Report 5-1.

OUTER JOIN Operator

To return part 1 and part 3 of the Venn diagram in Figure 3, we need to use two different variations of the *OUTER JOIN* operator. To return part 1, we need to use a *LEFT OUTER JOIN* operation, but to return part 3, we use a *RIGHT OUTER JOIN* operation. To demonstrate what is returned using the *LEFT OUTER JOIN* operations, I run the code in Listing 5-3.

Listing 5-3: Using a LEFT OUTER JOIN operation

```
SELECT [SET A].Member AS "[SET A] Member"
      ,[SET B].Member AS "[SET B] Member"
FROM [SET A] LEFT OUTER JOIN [SET B]
ON [SET A].Member = [SET B].Member;
```

When the code in Listing 5-3 is run, the output in Report 5-2 is returned.

Report 5-2: Output from basic *LEFT OUTER JOIN* operation

```
[SET A] Member [SET B] Member
-------------- --------------
1              NULL
1              NULL
2              NULL
3              NULL
4              NULL
4              NULL
5              5
5              5
6              6
6              6
```

When the *LEFT OUTER JOIN* operation of Listing 5-3 is performed, the members in *[SET A]* are matched to the members in *[SET B]* by using the column identified in the 'ON' clause. If a match is not found, then the value of the '[SET B] Member' column in the output is set to NULL. In my example, this happens when trying to match member values of 1 through 4 in *[Set A]* with *[SET B].* When a match is found between the members of *[SET A]* and *[SET B]* , then the member value of *[SET B]* is returned. This is why the output for members values of 5 and 6 in column '[SET A] Members' have matching values for the '[SET B] Member' column. In order to return only the members in To return part 1 of the Venn diagram in Figure 5-3, I have to add a *WHERE* predicate to the code in Listing 5-3, as can be found in Listing 5-4.

Listing 5-4: Returning Part 1 using a LEFT OUTER JOIN operator

```
SELECT [SET A].Member AS "[SET A] Member"
      ,[SET B].Member AS "[SET B] Member"
FROM [SET A] LEFT OUTER JOIN [SET B]
ON [SET A].Member = [SET B].Member
WHERE [SET B].Member IS NULL;
```

When the code in Listing 5-4 is run, the output in Report 5-3 is displayed. By looking at the '[SET A] Member' column in this output, you can see that only the members from *[SET A]* that are in part 1 of the joined set are returned.

Report 5-3: Part 1 output from LEFT OUTER JOIN operation with WHERE predicate

```
[SET A] Member [SET B] Member
------------- -------------
1             NULL
1             NULL
2             NULL
3             NULL
4             NULL
4             NULL
```

To get the results of part 3 of Figure 5-3, I just need to specify a *RIGHT OUTER JOIN* operation with a *WHERE* predicate, as shown in the code in Listing 5-5.

Listing 5-5: Returning Part 3 using a RIGHT OUTER JOIN operator

```
SELECT [SET A].Member AS "[SET A] Member"
      ,[SET B].Member AS "[SET B] MEMBER"
FROM [SET A] RIGHT OUTER JOIN [SET B]
ON [SET A].Member = [SET B].Member
WHERE [SET A].Member IS NULL;
```

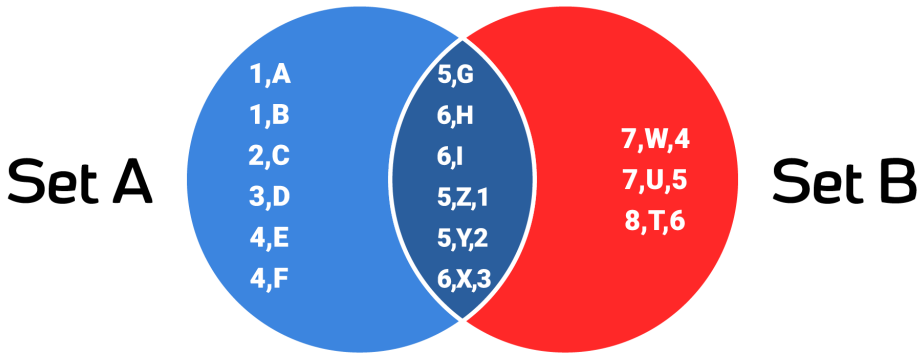
If you run this code, you will see that the member values of 7 and 8 are returned under the '[SET B] Member' column heading, but the values under '[SET A] Member' column heading contains the NULL values.

Joining two sets with Different Number of Columns

My examples above showed joining two sets that only have a single column, but the join operators can handle joining sets with multiple columns. It can even handle joining two sets that have different numbers of columns. To demonstrate this, I will create a couple of new tables *([SET C]* and *[SET D]*) that have multiple columns. The Venn diagram shown in Figure 5-4 represents the new tables I will be creating and illustrates how the two sets will be joined.

Figure 5-4: Sets with multiple columns

To create these two tables, I will use the code in Listing 5-6.



Listing 5-6: Creating sample tables with multiple columns

```
SET NOCOUNT ON;
-- Create Set C
DROP TABLE IF EXISTS [SET C];
CREATE TABLE [SET C] (Id_C TINYINT, Code CHAR(1));
INSERT INTO [SET C] VALUES (1,'A');
INSERT INTO [SET C] VALUES (1,'B');
INSERT INTO [SET C] VALUES (2,'C');
INSERT INTO [SET C] VALUES (3,'D');
INSERT INTO [SET C] VALUES (4,'E');
INSERT INTO [SET C] VALUES (4,'F');
INSERT INTO [SET C] VALUES (5,'G');
INSERT INTO [SET C] VALUES (6,'H');
INSERT INTO [SET C] VALUES (6,'I');
-- Create Set D
DROP TABLE IF EXISTS [SET D];
CREATE TABLE [SET D] (Id_D TINYINT, CodeX CHAR, CodeY TINYINT);
INSERT INTO [SET D] VALUES (5,'Z',1);
INSERT INTO [SET D] VALUES (5,'Y',2);
INSERT INTO [SET D] VALUES (6,'X',3);
INSERT INTO [SET D] VALUES (7,'W',4);
INSERT INTO [SET D] VALUES (7,'U',5);
INSERT INTO [SET D] VALUES (8,'T',6);
```

To demonstrate an *INNER JOIN* operation to return part 2 of Figure 5-4, I will run the code in Listing 5-7.

Listing 5-7: INNER JOIN operation on sets with different number of columns

```
SELECT * FROM [SET C] INNER JOIN [SET D]
        ON [SET C].Id_C = [SET D].Id_D;
```

When the code in Listing 5-7 is run, the output in Report 5-4 is returned. By reviewing the output in Report 5-4, you can see rows matched between *[SET C]* and *[SET D]* are based on member values of 5 and 6 which is similar to my previous example that produced Report 5-1. However, this time the other columns contained in *[Set C]* and *[Set D]* tables are also returned.

Report 5-4: Output of INNER JOIN operation of two sets with different number of columns

```
Id_C Code Id_D CodeX CodeY
---- ---- ---- ----- -----
5    G    5    Z     1
5    G    5    Y     2
6    H    6    X     3
6    I    6    X     3
```

If I were to modify the code in Listing 5-7 to perform the outer join operations, you would see that these additional columns are returned for the outer join queries as well. I will leave it up to you to build and run these outer join queries.

Set Theory and the Mathematics of Joining SQL Server Tables

When writing application code, you will join sets to return the data your applications need. Understanding set theory and the mathematics of joining SQL Server tables is a key part to writing good application code. You will find that the *INNER JOIN* and *OUTER JOIN* operators are probably the most frequently used operators you will use to join two sets when writing T-SQL code. Therefore, understanding how both JOIN operators work, along with the INTERSECT, UNION and EXCEPT operators, will allow you to quickly and easily write the code to return the data you need from your SQL Server databases.

Chapter 6

The Basics of Sorting Data Using the ORDER BY Clause

Back in Chapter 1, I showed you how to select rows from your SQL Server tables, and then in Chapters 4 and 5, I introduced you to joining tables and rows together using the INTERSECT, UNION, EXCEPT, and JOIN operators. In those articles, the SELECT statement examples all returned data in an arbitrary order. Sometimes you will need your returned row set sorted in a particular order so you can more easily examine the results looking for specific data. In this chapter, I will show you how to use the ORDER BY clause to return your data in sorted order.

Ordering your records

When you add the ORDER BY clause to your SELECT statement, SQL Server usually uses the SORT operator to organize the returned result set. If you happen to be sorting your result set by the clustered index key, then a SORT operator may not be needed. You can use the *ORDER BY* clause to sort your data in ascending or descending order based on a column, a set of columns, or an expression. The ORDER BY clause is not valid in views, inline table-valued functions, derived tables, or sub-queries unless you also use the TOP clause in your statement.

Listing 6-1 shows the syntax for the *ORDER BY clause:*

Listing 6-1: The ORDER BY clause syntax

```
ORDER BY
order_by_expression
[COLLATE collation_name]
[ASC | DESC][,…N]
```

Where:

- *order_by_expression* identifies the sort criteria. The sort criteria can be a column name, column alias name, an integer that represents the position of the column in the selection list, or an expression.
- *collation_name* identified the collation that will be used when sorting on a specified column. If no *COLLATE* parameter is used, then the column is sorted based on the collation associated with the columns used in the *order_by_expression*.
- ASC or DESC represents the way the *order_by_expression* will be sorted. ASC stands for ascending sequence, and DESC stands for descending sequence. If neither ASC nor DESC is specified, then the *order_by_expression* is sorted in ascending sequence.

To demonstrate how to use the ORDER BY clause, I first will create a couple of tables and then populate them with several rows of data. I will then execute several different SELECT statements that contain different *ORDER BY clauses* to retrieve rows from these tables. If you want to follow along and execute the SELECT statements in this article, you can create my sample tables by running the code in Listing 6-2.

Listing 6-2: Script to create sample tables

```
USE tempdb;
GO
SET NOCOUNT ON;
-- Create sample tables
-- Create table Car
CREATE TABLE [dbo].[Car](
   [ID] [int] IDENTITY(1,1) NOT NULL,
   [Make] [varchar](50) NULL,
   [Model] [varchar](50) NULL,
   [NumOfDoors] [tinyint] NULL,
   [ModelYear] [smallint] NULL,
   [ColorId] [int] NULL
) ON [PRIMARY]
-- Create Table Color
CREATE TABLE [dbo].[Color](
   [ID] [int] IDENTITY(1,1) NOT NULL,
   [Color] [varchar](50) NULL
) ON [PRIMARY];
-- Populate Color table
INSERT INTO Color VALUES('Red');
INSERT INTO Color VALUES('White');
INSERT INTO Color VALUES('Blue');
INSERT INTO Color VALUES('Green');
-- Populate Car Table
INSERT INTO Car VALUES ('Ford','Mustang','2','1964',1);
INSERT INTO Car VALUES ('ford','F150','2','2010',1);
INSERT INTO Car VALUES ('Toyota','Camry','4','2011',1);
INSERT INTO Car VALUES ('Ford','Taurus','5','1995',2);
INSERT INTO Car VALUES ('ford','F250','2','2010',3);
INSERT INTO Car VALUES ('Chevrolet','Volt','4','2010',1);
INSERT INTO Car VALUES ('Ford','Focus','4','2012',4);
```

```
INSERT INTO Car VALUES ('Chevrolet' ,'Aveo','4','2011',2);
INSERT INTO Car VALUES ('Chevrolet','Camaro','4','1978',4);
INSERT INTO Car VALUES ('Honda','Civic','4','2012',1);
INSERT INTO Car VALUES ('Chevrolet','Cruse','4','2012',1);
INSERT INTO Car VALUES ('Toyota','Rav4','5','2000',1);
```

This script creates two tables, *Car* and *Color*, in the *tempdb* database. Each of my examples in this article will use one or both of these tables to demonstrate how to use the ORDER BY to sort your result set.

Ordering by a Single Column

A basic SELECT statement without an ORDER BY clause will return records in an arbitrary order but might actually be ordered if an index was used to retrieve the records. However, we can never be sure of the order that the rows will be returned without ORDER BY. If I need my results set to be returned in a specific order, I need to add the ORDER BY clause to my SELECT statement and specify the column or columns I want SQL Server to sort on. For my first example, found in Listing 6-3, I will sort my data on *Car.Make* in ascending sequence.

Listing 6-3: *SELECT* statement that orders data by *Car.Make*

```
USE tempdb;
GO
SELECT Car.Make
   ,Car.Model
   ,Car.NumOfDoors
   ,Car.ModelYear
   ,Color.Color
FROM Car INNER JOIN Color
  ON Car.ColorId = Color.ID
ORDER BY Car.Make ASC;
```

When I run the code in Listing 6-3, I get the output shown in Report 6-1. By reviewing this report, you can see that all the rows are returned in alphabetic or ascending order based on the *Car.Make* column.

Report 6-1: Output when ordering by *Car.Make*

```
Make                          Model        NumOfDoors ModelYear Color
----------------------------- ------------ ---------- --------- ----------
Chevrolet                     Volt         4          2010      Red
Chevrolet                     Aveo         4          2011      White
Chevrolet                     Camaro       4          1978      Green
Chevrolet                     Cruse        4          2012      Red
Ford                          Focus        4          2012      Green
Ford                          Mustang      2          1964      Red
ford                          F150         2          2010      Red
Ford                          Taurus       5          1995      White
ford                          F250         2          2010      Blue
Honda                         Civic        4          2012      Red
Toyota                        Camry        4          2011      Red
Toyota                        Rav4         5          2000      Red
```

Notice that I didn't specify the ASC or DESC in my ORDER BY specification. By not specifying the sort direction, SQL Server will sort in ascending (ASC) order.

Sorting Data based on collation

If you look closely at the output in Report 6-1, you will notice that the first character in the *Make* column has both upper and lower case spelling for the value "Ford". Since the *Make* column in table *Car* has a collation setting of case insensitive in my *tempdb* database, the upper and lower case spellings of "Ford" are interleaved in the output. This happened because the ORDER BY clause follows the sorting rules associated with collation settings of columns specified in the ORDER BY clause. If I want to have my

output sorted based on case, so all the upper case Fords would be together, and the lower case fords would be next to each other, I would need to use the COLLATE options associated with the *ORDER BY* clause. In Listing 6-4, I have a SELECT statement that uses the COLLATE clause as part of the *ORDER BY* specification. This *SELECT* statement will return the same rows shown in Report 6-1, but the returned rows will now be sorted based on case.

Listing 6-4: *SELECT* statement that orders data based on case sensitivity by *Car.Make*

```
USE tempdb;
GO
SELECT Car.Make
   ,Car.Model
   ,Car.NumOfDoors
   ,Car.ModelYear
   ,Color.Color
FROM Car INNER JOIN Color
  ON Car.ColorId = Color.ID
  ORDER BY Car.Make
  COLLATE SQL_Latin1_General_CP1_CS_AS ASC;
```

I'll leave it up to you to run the SELECT statement in Listing 6-4 to show how the cars with a make of "Ford" will be sorted based on case.

Sorting Data in Descending Sequence

There are times you want to see records sorted in an order where the last item alphabetically or numerically appears first in the result set. To do this, you need to use the DESC (descending) option of the ORDER BY clause. If I want to see the newest car by *ModelYear* at the top of my result set, I could order my result set in descending sequence by *Model Year*. This is what I have done in Listing 6-5.

Listing 6-5: Order by *Car.ModelYear* in descending order.

```
USE tempdb;
GO
SELECT Car.Make
   ,Car.Model
   ,Car.NumOfDoors
   ,Car.ModelYear
   ,Color.Color
FROM Car INNER JOIN Color
  ON Car.ColorId = Color.ID
ORDER BY Car.ModelYear DESC;
```

When I run the code in Listing 6-5, I get the output in Report 6-2.

Report 6-2: Output in descending order by *Car.ModelYear*

```
Make         Model       NumOfDoors ModelYear Color
------------ ----------- ---------- --------- --------------------
Ford         Focus       4          2012      Green
Honda        Civic       4          2012      Red
Chevrolet    Cruse       4          2012      Red
Chevrolet    Aveo        4          2011      White
Toyota       Camry       4          2011      Red
ford         F150        2          2010      Red
ford         F250        2          2010      Blue
Chevrolet    Volt        4          2010      Red
Toyota       Rav4        5          2000      Red
Ford         Taurus      5          1995      White
Chevrolet    Camaro      4          1978      Green
Ford         Mustang     2          1964      Red
```

By reviewing this output, you can see the cars with *ModelYear* of 2012 are at the top of the list of in my results.

Sorting Data Based on Column Not in the SELECT List

SQL allows you to sort your result set on columns that are not in the SELECT list. To demonstrate this, I have created the code in Listing 6-6.

Listing 6-6: Order by *Car.ModelYear* in descending order

```
USE tempdb;
GO
SELECT Make
   ,Model
FROM Car
ORDER BY Car.ModelYear DESC;
```

This code, when run, will return the *Make* and *Model* columns sorted by *Car.ModelYear*, even though the *Car.ModelYear* is not included in the SELECT list.

Sorting Data based on Column Ordinal Position

Another option for specifying which column to sort on is to use a column ordinal position. By ordinal position, I mean the order in which the columns appear in the SELECT list. One reason that you might use the ordinal position in your ORDER BY clause is when the select list column you want to sort on contains a complicated expression, such as a CASE expression. You also need to keep in mind when you do use an ordinal position, that adding additional columns to your select list might require changing the ordinal positions in the ORDER BY clause. If you don't change the ordinal position value when adding columns to your select list, you might get your result set sorted incorrectly. Listing 6-7 contains a SELECT statement that uses a column ordinal position for the *Car.ModelYear* column in the ORDER BY clause.

Listing 6-7: Ordering data using a column reference number

```
USE tempdb;
GO
SELECT Car.Make
   ,Car.Model
   ,Car.NumOfDoors
   ,Car.ModelYear
   ,Color.Color
FROM Car INNER JOIN Color
  ON Car.ColorId = Color.ID
ORDER BY 4 DESC;
```

When Listing 6-7 is run, it will return a sorted result set identical the results in Report 6-2.

Sorting Data based on Column Alias

There may be times when one or more of the columns in the selection list of a SELECT statement has alias names associated with it. A couple of reasons why an alias names might be used is because either the column in the selection list doesn't have a column name associated with it, or you want the output to contain a column heading with a different name than the original column name. If a SELECT query has an alias name then the output can be sorted using the alias name, as is shown in Listing 6-8

Listing 6-8: Ordering data using an alias name

```
USE tempdb;
GO
SELECT Car.Make + '_ ' + Car.Model
   ,Car.NumOfDoors
   ,Car.ModelYear
   ,Color.Color
FROM Car INNER JOIN Color
```

```
  ON Car.ColorId = Color.ID
ORDER BY Make_Model DESC;
```

When Listing 6-8 is run, it get the results in Report 6-3.

Report 6-3: Output in descending order by *Make_Model*

```
Make_Model          NumOfDoors   ModelYear   Color
-----------------   ----------   ---------   -----
Toyota_Rav4         5            2000        Red
Toyota_Camry        4            2011        Red
Honda_Civic         4            2012        Red
Ford_Taurus         5            1995        White
Ford_Mustang        2            1964        Red
Ford_Focus          4            2012        Green
ford_F250           2            2010        Blue
ford_F150           2            2010        Red
Chevrolet_Volt      4            2010        Red
Chevrolet_Cruse     4            2012        Red
Chevrolet_Camaro    4            1978        Green
Chevrolet_Aveo      4            2011        White
```

If you look at the column headings in Report 6-3 you will see that the report is ordered by the alias name *Make_Model* in descending sequence.

Sorting Data Based on Multiple Columns

Each of my examples so far has shown you how to order a result set based on a single column. The ORDER BY clause allows you to order record sets based on multiple columns or expressions. You can even specify a different sort direction (ascending, or descending) for each column specified. In Listing 6-9, I have a SELECT statement that will sort the results based on *Car.Make* in ascending order, and the *Car.ModelYear* in descending order.

Listing 6-9: Ordering data using multiple columns

```
USE tempdb;
GO
SELECT Car.Make
   ,Car.Model
   ,Car.NumOfDoors
   ,Car.ModelYear
   ,Color.Color
FROM Car INNER JOIN Color
  ON Car.ColorId = Color.ID
ORDER BY Car.Make ASC
        , Car.ModelYear DESC;
```

Numeric Character Data Sorted Alphabetically

When sorting numeric data that is contained in character data type columns, the data will be sorted alphabetically. Since data is sorted alphabetically, this means each column will be sorted based on the characters in the column from left to right. This means the number 10 will sort before the number 2 because the first digit (1) of the number 10 comes alphabetically before the digit 2. The sample code in Listing 6-10 demonstrates this behavior.

Listing 6-10: Sorting numeric data stored in a varchar column

```
USE tempdb;
GO
CREATE TABLE NumericTest (Name VARCHAR(20), Number VARCHAR(4));
INSERT INTO NumericTest VALUES ('One','1');
INSERT INTO NumericTest VALUES ('One Thousand','1000');
INSERT INTO NumericTest VALUES ('Two','2');
INSERT INTO NumericTest VALUES ('Twenty' ,'20');
INSERT INTO NumericTest VALUES('Ten','10');
SELECT * FROM NumericTest
ORDER BY Number;
DROP TABLE NumericTest;
```

When this code is run the output in Report 6-4 is produced. If you look at this output, you will see that 10, and 1000 sort before 2 and 20.

Report 6-4: Numeric Data Sorted Alphabetically

```
Name                 Number
-------------------  ------
One                  1
Ten                  10
One Thousand         1000
Two                  2
Twenty               20
```

Since the *Number* column is defined as a varchar(4), SQL Server sorted this column alphabetically, even though this column only contains numbers. If the *Number* column had been defined using a numeric data type, SQL Server would have sorted these records based on the numeric values in this column. This would have caused 2 to come before 10, and 1000 when sorted in ascending sequence. If you do happen to have numeric data stored in a

character field type, you can use the trick shown in Listing 6-11 to resolve the sorting issue demonstrated by Listing 6-10.

Listing 6-11: Sorting numeric data stored in a varchar column

```
USE tempdb;
GO
CREATE TABLE NumericTest (Name VARCHAR(20), Number VARCHAR(4));
INSERT INTO NumericTest VALUES ('One','1');
INSERT INTO NumericTest VALUES ('One Thousand','1000');
INSERT INTO NumericTest VALUES ('Two','2');
INSERT INTO NumericTest VALUES ('Twenty' ,'20');
INSERT INTO NumericTest VALUES('Ten','10');
SELECT * FROM NumericTest
ORDER BY CAST(Number AS INT);
DROP TABLE NumericTest;
```

In Listing 6-11, I used the CAST function in the ORDER BY clause to convert the character data to an integer prior to SQL Server sorting the rows. If you run this code, you will see that the numbers are now sorted according to their numeric value. This code also demonstrates how you can use an expression or function in your ORDER BY clause.

Create Order to Your Selected Data

Using the basic SELECT statement returns your data in an arbitrary order. To guarantee that your data will be returned by SQL Server in a specific order, you need to have an ORDER BY clause. The ORDER BY clause allows you to sort your data by a single column or multiple columns, in descending or ascending order. SQL Server also allows you to use functions or expressions in the ORDER BY clause so calculated values can be used to order your data. Add order to your selected data by using the ORDER BY clause.

Chapter 7

Using Logical Operators

SQL Server supports a number of different logical operators. These operators can be used for testing Boolean conditions that return true, false and unknown in your T-SQL code. Logical operators are useful for defining constraints to limit the rows to be processed when selecting or updating data. This chapter will provide an overview of the logical operators supported by SQL Server and a few examples for some of those commonly used operators.

Logical Operators

Logical operators are one of several types of operators supported by SQL Server. Logical operators are used to test for TRUE, FALSE, or UNKNOWN conditions. Table 7-1 has a list of the logical operators that can be found in the Microsoft documentation.

Table 7-1: Logical Operators

Operator	Meaning
ALL	TRUE if all of a set of comparisons are TRUE.
AND	TRUE if both Boolean expressions are TRUE.
ANY	TRUE if any one of a set of comparisons is TRUE.
BETWEEN	TRUE if the operand is within a range.
EXISTS	TRUE if a subquery contains any rows.

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.
SOME	TRUE if some of a set of comparisons are TRUE.

These operators can be used by themselves or together with other operators in predicates to help build the appropriate business constraints when maintaining, managing, and reporting on data. In order to understand how to use logical operators, I will provide a few examples that demonstrate how to use the BETWEEN, LIKE and IN logical operators to solve different processing requirements. I will use the AdventureWorks2017 database for my examples.

BETWEEN operator

The BETWEEN operator allows you to test if an expression is within a specific range of values. The syntax for the BETWEEN operator, as shown in the Microsoft documentation, can be found in Figure 7-1.

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

test_expression
Is the expression to test for in the range defined by *begin_expression* and *end_expression*. *test_expression* must be the same data type as both *begin_expression* and *end_expression*.

NOT
Specifies that the result of the predicate be negated.

begin_expression
Is any valid expression. *begin_expression* must be the same data type as both *test_expression* and *end_expression*.

end_expression
Is any valid expression. *end_expression* must be the same data type as both *test_expression*and *begin_expression*.

AND
Acts as a placeholder that indicates *test_expression* should be within the range indicated by *begin_expression* and *end_expression*.

Figure 7-1: Syntax of the BETWEEN operator

The BETWEEN operator will return a value of TRUE when expression being tested is greater than or equal to the beginning value and less than or equal to the ending value specified. If the expression being tested is not within the range, then FALSE will be returned. If any of the input to the BETWEEN operator is NULL, then UNKNOWN is returned.

To show how the BETWEEN logical operator works, let's assume you want a report that shows the number of orders placed each date in January 2012. To meet this reporting requirement, I will use the code in Listing 7-1.

Listing 7-1: Using BETWEEN operator

```
USE AdventureWorks2017;
GO
SELECT DISTINCT CONVERT(char(10),OrderDate,121) AS [OrderDateYYYY-MM-DD]
      ,COUNT(*) AS NumOfOrders
FROM Sales.SalesOrderHeader
WHERE OrderDate Between '2011-06-01'
                    and '2011-06-30'
GROUP BY OrderDate
ORDER BY [OrderDateYYYY-MM-DD];
```

The code in Listing 1 contains a SELECT statement that returns all order dates, in YYYY-MM-DD format and the number of orders placed, where the OrderDate values from the SalesOrderHeader table are between the value “2012-01-01” and “2012-12-31”.

When I run the code in Listing 7-1, I get the results in Report 7-1.

Report 7-1: Results when code in Listing 7-1 is run.

```
OrderDateYYYY-MM-DD   NumOfOrders
-------------------   -----------
2011-06-01            4
2011-06-02            5
2011-06-03            2
…
2011-06-28            5
2011-06-29            5
2011-06-30            5
```

This example shows how the BETWEEN operator returns an inclusive range. Meaning it returns all values in the range, including the endpoint values. If you want to specify an exclusive range that doesn’t consider the endpoint values, then you will need to use the greater than (>) and/or less than (<) operators.

LIKE Operator

There may be a time when you want to find all the records in a table where a column contains a specific string somewhere within the column value. Suppose you want to find all the last names that end in “sen”. If you have this kind of selection criteria, then the LIKE operator can be used to find those records. The syntax for the LIKE operator as shown in the Microsoft documentation can be found in Figure 7-2.

match_expression [NOT] LIKE pattern [ESCAPE escape_character]

match_expression
Is any valid expression of character data type.

pattern
Is the specific string of characters to search for in *match_expression*, and can include the following valid wildcard characters. *pattern* can be a maximum of 8,000 bytes.

Wildcard character	Description	Example
%	Any string of zero or more characters.	WHERE title LIKE '%computer%' finds all book titles with the word 'computer' anywhere in the book title.
_ (underscore)	Any single character.	WHERE au_fname LIKE '_ean' finds all four-letter first names that end with ean (Dean, Sean, and so on).
[ ]	Any single character within the specified range ([a-f]) or set ([abcdef]).	WHERE au_lname LIKE '[C-P]arsen' finds author last names ending with arsen and starting with any single character between C and P, for example Carsen, Larsen, Karsen, and so on. In range searches, the characters included in the range may vary depending on the sorting rules of the collation.
[^]	Any single character not within the specified range ([^a-f]) or set ([^abcdef]).	WHERE au_lname LIKE 'de[^l]%' all author last names starting with de and where the following letter isn't l.

escape_character
Is a character put in front of a wildcard character to indicate that the wildcard is interpreted as a regular character and not as a wildcard. *escape_character* is a character expression that has no default and must evaluate to only one character.

Figure 7-2: Syntax for the LIKE operator

As you can see from the syntax in Figure 7-2, to use the LIKE operator, you need to identify a pattern to search for. That pattern can be the exact string you want to find, or a pattern can use wild cards if you want to search for different, but many similar strings.

To show how the LIKE operator works, I will search the LastName column of the Person.Person table, looking for any LastName that starts with a "C" or an "H", and ends in "sen". I will use the code in Listing 7-2 to find the rows that meet these search criteria.

Listing 7-2: Searching Last Names that match a pattern

```
USE AdventureWorks2017
GO
SELECT FirstName, MiddleName, LastName
FROM  Person.Person
WHERE LastName LIKE '[CH]%sen';
```

In Listing 7-2, I used a pattern value of '[CH]%sen' to find those last names. The "[CH]" portion of the pattern, uses the single character wildcard specification to find those LastName's that start with "C" or "H". To find just those LastName that ends in "sen" I use the wildcard specification of "%sen". This specification uses the "%" sign to match any string of characters, whereas the "sen" portion is used to identify the last name must end in "sen". When I run the code in Listing 7-2, I get the output found in Report 7-2.

Report 7-2: Records returned after running code in Listing 7-2

```
FirstName   MiddleName   LastName
---------   ----------   --------
Charles     M.           Christensen
Ryan        NULL         Cornelsen
Ryan        L            Cornelsen
Jay         NULL         Henningsen
```

The power of wildcarding within the LIKE operator makes it easy to search for any pattern.

IN Operator

There may be a time when you need to find rows where a given column contains one or more different values. In this situation, the IN operator will be helpful. The syntax, as found in the Microsoft documentation for the IN operator, can be found in Figure 7-3.

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

test_expression
Is any valid expression.

subquery
Is a subquery that has a result set of one column. This column must have the same data type as *test_expression*.

expression[,... *n*]
Is a list of expressions to test for a match. All expressions must be of the same type as *test_expression*.

Figure 7-3: IN operator syntax

As you can see, the IN operator accepts one or more subqueries or an expression to identify the values you want to find or not find. In order to show you how this works, I will first show a simple example that uses the IN operator to look for three different possible values. The code for this example can be found in Listing 7-3.

Listing 7-3: Simple IN operator example

```
USE AdventureWorks2017
GO
SELECT Name, ListPrice
FROM  Production.Product
WHERE Name IN ('Road Tire Tube', 'Touring Pedal', 'Minipump');
```

By reviewing the code in Listing 7-3, you can see I'm using the IN operator to search the Production.Product.Name column for three specific values. When I run this code, I get the output in Report 7-3.

Report 7-3: Output when code in Listing 7-3 is executed

```
Name              ListPrice
--------------    -----------
Minipump          19.99
Road Tyre Tube    3.99
Touring Pedal     80.99
```

Another common way to identify the values for the IN operator is using a subquery. By using a subquery, the values can be selected based on simple, or complicated TSQL code. Suppose the boss asks for a query to be written for a special promotion the company is running. His criteria for this query, is to produce a list of all customers who have ever purchased a product that had the 4 characters "bike" in the product name. The SELECT statement in Listing 7-4 meets the promotional reporting requirements.

Listing 7-4: Using multiple query in the IN operator

```
SELECT  DISTINCT TOP 5 H.CustomerId FROM Sales.SalesOrderHeader AS H
   INNER JOIN Sales.SalesOrderDetail AS D
   ON H.SalesOrderID = D.SalesOrderID
   WHERE D.ProductID  IN (
      SELECT ProductID FROM
      Production.Product
      WHERE Name LIKE '%bike%')
ORDER BY H.CustomerID;
```

The code in Listing 7-4 uses a subquery within the IN operator to identify the top 5 CustomerID values that meet the bosses promotional criteria. When the code in Listing 7-4 is run, it returns the results in Report 7-4.

Report 7-4: Output when code in Listing 7-4 is executed

```
CustomerID
----------
11015
11019
11024
11039
11046
```

One thing worth mentioning is when you use a single subquery with the IN operator the subquery must reference only a single column name, but the query can return multiple values.

Pitfalls of Using Logical Operators

You have already seen how the IN operator has some issues when multiple subqueries are used. There are a few other pitfalls worth mentioning. The first one is dealing with NULL values.

A NULL value in a column means the column doesn't have a value. Therefore, when using logical operators to search for columns that contain NULL, they can't be found using the code in Listing 7-5.

Listing 7-5: Searching for Nulls

```
USE AdventureWorks2017
GO
SELECT Name, Color FROM Production.Product
WHERE Color IN ('White', 'Grey', NULL);
```

The code in Listing 7-5 will not return any products that have a NULL value for the Color column. If you want to find the Production.Products that are "White", "Grey", or NULL, then you have to convert those NULL values to a non-null value prior to comparing it using the IN logical operator. The code in Listing 7-6 will find all the products that have a NULL value for the Color attribute.

Listing 7-6: Finding NULL Colors

```
USE AdventureWorks2017
GO
SELECT Name, Color FROM Production.Product
WHERE COALESCE(Color,'') IN ('White', 'Grey', '');
```

In Listing 7-6, I am using the COALESCE function to convert NULL values in the Color column to the empty string so it could be compared to values in the IN operator.

Does case matter when you use a logical operator? The answer is “It Depends”. If the column being proceeded with a logical operator contains upper and lower case characters, and your database and/or column is not case sensitive, then you will need to do some coding to find those case sensitive values. To demonstrate how case sensitivity may or may not affect the results of a query, consider the code in Listing 7-7.

Listing 7-7: Code to show case-sensitivity issue

```
USE AdventureWorks2017
GO
DECLARE @Colors TABLE (Color varchar(15)) ;
INSERT INTO @Colors VALUES ('White'),
                           ('white'),
                               ('Grey'),
                                ('grey');
SELECT Color from @Colors
WHERE Color IN ('white','grey');
```

When I run the code in Listing 7-7, I get the results in Report 7-5.

Report 7-5: Results when executing code in Listing 7-7

```
Color
--------
White
white
Grey
grey
```

The code in Listing 7-7 returned all the different colors regardless of the case of each character in the Color column. Because the database AdventureWorks2017 is case insensitive, the IN operator matches both cases of the color white and grey. If I only wanted

to return the lower case white and grey colors, I could use the COLLATE cause as I did in Listing 7-8.

Listing 7-8: Dealing with Case-Sensitive values

```
USE AdventureWorks2017
GO
DECLARE @Colors TABLE (Color varchar(15)) ;
INSERT INTO @Colors VALUES ('White'),
                           ('white'),
                           ('Grey'),
                           ('grey');
SELECT Color from @Colors
WHERE Color COLLATE Latin1_General_CS_AS IN ('white','grey');
```

In Listing 7-8, the COLLATE clause to make SQL Server treat the Color column as if it's case-sensitive, so the comparison can be performed as if the column was case-sensative. By doing this conversion, I can use the IN operator to perform case-sensitive searches.

Using Logical Operators

Using Logical Operators allow some additional ways to constrain and join SQL Server data using Boolean operations that equate to TRUE, FALSE or UNKNOWN. These operators allow you not only specify characters string for your search criteria, but also allow you to use wildcards and subqueries to identify those search values. In this chapter, I only covered a few logical operators. I suggest you review the documentation to fully understand all the different logical operators available in SQL Server.

Chapter 8

Using Scalar Functions

SQL Server comes with a number of different types of built-in functions. One of these types of functions is called “Scalar”. A scalar function is a function that when called returns only a single value.

The other category of built-in functions are aggregate, analytic, and rowset, functions. Some information about these other types of function can be found in other chapters of this book. For a complete reference to all the different types of built-in functions within SQL Server, please refer to the Microsoft documentation: https://docs.microsoft.com/en-us/sql/tsql/functions/functions?view=sql-server-ver15.

In order to understand scalar functions, I will explore the different categories of scalar functions first. I will follow up by providing a few examples of how to use some of the more commonly used scalar functions. Let’s dig into scalar functions in detail.

Categories of Scalar Functions

As already stated, a scalar function is a function that returns only a single value. They might require no parameters or may require one or more parameters. Scalar functions can be used anywhere an expression can be used. There are a lot of built-in scalar functions in SQL Server, too many to list in this chapter. Each function will fall into one of the following sub-categories of scalar functions listed in Table 8-1.

Table 8-1: Subcategories of scalar functions

Configuration	Metadata
Conversion	Security
Cursor functions	String
Date and Time	Security
JSON	System
Logical	System Statistical
Mathematical	Text and image

Determinism of a Function

Prior to getting into examples of how to use scalar functions, let me first discuss an important concept about functions, known as determinism. The determinism of a function can be either deterministic or non-deterministic. A deterministic function is a function that, when called with the same input value multiple times, will return the same output value each time it's called. Whereas a non-deterministic function, when called numerous times with the same values, may or may not return the same results. For example, if you call the deterministic string function LEN and pass it a parameter value of "This is my string.", it will always return the integer value "18", because that is the length of the string. Whereas if you call the non-deterministic Date/Time function GETDATE() which excepts no parameter, it will always return a different value, the current date and time. In Table 8-2, you can find a list of built-in deterministic functions.

Table 8-2: List of Built-In Deterministic Functions

ABS	DATEDIFF	POWER
ACOS	DAY	RADIANS
ASIN	DEGREES	ROUND
ATAN	EXP	SIGN
ATN2	FLOOR	SIN
CEILING	ISNULL	SQUARE
COALESCE	ISNUMERIC	SQRT
COS	LOG	TAN
COT	LOG10	YEAR
DATALENGTH	MONTH	
DATEADD	NULLIF	

Whereas in Table 8-3, you can find a list of built-in non-deterministic functions.

Table 8-3: List of Non-Deterministic Built-In Functions

@@CONNECTIONS	GETDATE
@@CPU_BUSY	GETUTCDATE
@@DBTS	GET_TRANSMISSION_STATUS
@@IDLE	LAG
@@IO_BUSY	LAST_VALUE
@@MAX_CONNECTIONS	LEAD
@@PACK_RECEIVED	MIN_ACTIVE_ROWVERSION
@@PACK_SENT	NEWID
@@PACKET_ERRORS	NEWSEQUENTIALID
@@TIMETICKS	NEXT VALUE FOR
@@TOTAL_ERRORS	NTILE
@@TOTAL_READ	PARSENAME
@@TOTAL_WRITE	PERCENTILE_CONT
AT TIME ZONE	PERCENTILE_DISC
CUME_DIST	PERCENT_RANK

CURRENT_TIMESTAMP	RAND
DENSE_RANK	RANK
FIRST_VALUE	ROW_NUMBER
FORMAT	TEXTPTR

Database Used for Examples

In order to demo some of the different built-in scalar functions, I will need a database for testing my sample queries. The database I will be using for my examples will be the SQL Server sample database known as AdventureWorks2017.

Examples of using Scalar functions

A Scalar function is a function that returns a single value. There are a number of different built-in scalar functions, as shown in "Scalar Functions" section above. In order to show how a scalar function works and how expressions can be pass as parameters to a function, I will be providing several examples.

Date/Time Function Examples

Date/Time functions allow you to generate and manipulate date/time values in SQL Server. For instance, you might just want to know the current date and time. Or maybe you want to calculate the difference between two dates or validate that a string value contains a valid date. These are just a few examples of how you might use a date/time function in your TSQL code. For a complete list of date/time functions review the Microsoft date and time documentation: https://docs.microsoft.com/en-us/sql/t-sql/functions/date-and-time-data-types-and-functions-transact-sql?view=sql-server-ver15. To get a better sense of how to use date/time functions, I will provide a few examples of some commonly used date/time functions.

Using the GETDATE() Function

There are many different situations where your applications might need to know "What is the current date and time?". The built-in GETDATE() function returns the current system timestamp as a datetime value without a time zone offset. The GETDATE() function does not require any parameters, as shown in the Microsoft Documentation https://docs.microsoft.com/en-us/sql/t-sql/functions/getdate-transact-sql?view=sql-server-ver15. The code in Listing 8-1 shows how to use the GETDATE() function to display the current date/time.

Listing 8-1: Returning the current date/time

```
SELECT GETDATE() AS CurrentDateTime;
```

As you can see here, I called the GETDATE() function without passing any parameters. When I ran the code in Listing 8-1, it produced the results found in Report 8-1.

Report 8-1: Results when executing code in Listing 8-1

```
CurrentDateTime
-----------------------
2020-04-14  09:20:27.953
```

By reviewing the results in Report 8-1, you can see that the current date/time was displayed as a DATETIME data type value.

Using the DATEDIFF function

There may be times when an application needs to compare two dates to determine how far apart the dates might be. The DATEDIFF functions can be used to return a count of the number of specific date part boundaries crossed between two different date/time values.

The syntax for the DATEDIFF function as found in the Microsoft Documentation can is displayed in Figure 8-1.

DATEDIFF (datepart , startdate , enddate)

datepart
The units in which **DATEDIFF** reports the difference between the *startdate* and *enddate*. Commonly used *datepart* units include month or second.

The *datepart* value cannot be specified in a variable, nor as a quoted string like 'month'.

The following table lists all the valid *datepart* values. **DATEDIFF** accepts either the full name of the *datepart*, or any listed abbreviation of the full name.

datepart name	*datepart* abbreviation
Year	yy, yyyy
quarter	qq, q
Month	mm, m
dayofyear	dy, y
Day	dd, d
Week	wk, ww
Hour	Hh
minute	mi, n
second	ss, s
millisecond	Ms
microsecond	Mcs
nanosecond	Ns

startdate
An expression that can resolve to one of the following values:

- date

- datetime
- datetimeoffset
- datetime2
- smalldatetime
- time

enddate
See *startdate*.

Figure 8-1: Syntax for DATEDIFF function

To better understand how to use the DATEDIFF function, I will run some code that will find all the OrderDate's, in the SalesOrderHeader table, that were placed in the first 65 days of the second quarter of 2011. The code for this example can be found in Listing 8-2.

Listing 8-2: DATEDIFF function example

```
USE AdventureWorks2017;
GO
SELECT DISTINCT(OrderDate)
                , DATEDIFF(day, DATEFROMPARTS('2011','04','01'),
OrderDate) as DaysBetween
FROM Sales.SalesOrderHeader
WHERE DATEDIFF(day, DATEFROMPARTS('2011','04','01'), OrderDate) < 65
ORDER BY OrderDate;
```

By reviewing the code in Listing 8-2, you can see that I used the DATEDIFF function not only in the select list but also in the WHERE clause. I also used an expression for the second parameter of this function. The expression used was the DATEFROMPARTS function, which is another scalar date function. I used this function to specify the startdate parameter. The DATEFROMPARTS function uses three different parameters values (year, month, and day) that are used to construct a DATE type value.

For more information on the DATEFROMPARTS function, please review the Microsoft document https://docs.microsoft.com/en-us/sql/t-sql/functions/datetimefromparts-transact-sql?view=sql-server-ver15. In this example, I am displaying all of the SalesOrderHeader rows where the difference between the startdate and enddate parameters of the DATEDIFF functions are less than 65 from the start date of the second quarter in 2011.

When the code in Listing 8-2 is run, the results found in Report 8-2 are returned.

Report 8-2: Results from running code in Listing 8-2

```
OrderDate                DaysBetween
-----------------------  -----------
2011-05-31 00:00:00.000  60
2011-06-01 00:00:00.000  61
2011-06-02 00:00:00.000  62
2011-06-03 00:00:00.000  63
2011-06-04 00:00:00.000  64
```

As you can see, the DATEDIFF function made it very easy for me to calculate the number of days, as an integer value, between two different dates.

Additionally, in this example, I used an expression for the startdate parameter, in this cause another function call. You can use expressions as parameters to functions as long as the expression equates to a value that has an appropriate data type for the function parameter.

Using the DATEADD function

There may be a time when you need to take a date and add or subtract a number of date units from it to derive another date. The DATEADD function performs those kinds of date math calculations. The syntax for the DATEADD function, as found in the Microsoft documentation, is shown in Figure 8-2.

DATEADD (datepart , number , date)

datepart

The part of *date* to which DATEADD adds an **integer** *number*. This table lists all valid *datepart* arguments.

Note

DATEADD does not accept user-defined variable equivalents for the *datepart* arguments.

datepart	Abbreviations
Year	**yy, yyyy**
quarter	**qq, q**
Month	**mm, m**
dayofyear	**dy, y**
Day	**dd, d**
Week	**wk, ww**
weekday	**dw, w**
Hour	**hh**
minute	**mi, n**
second	**ss, s**
millisecond	**ms**
microsecond	**mcs**
nanosecond	**ns**

number

An expression that can resolve to an int that DATEADD adds to a *datepart* of *date*. DATEADD accepts user-defined variable values for *number*. DATEADD will

truncate a specified *number* value that has a decimal fraction. It will not round the *number* value in this situation.

date
An expression that can resolve to one of the following values:

- **date**
- **datetime**
- **datetimeoffset**
- **datetime2**
- **smalldatetime**
- **time**

For *date*, DATEADD will accept a column expression, expression, string literal, or user-defined variable. A string literal value must resolve to a **datetime**.

Figure 8-2 Syntax for DATEADD function

To show an example of how to use this function, let me calculate the anticipated retirement vesting date, assuming it takes five years for an employee to become vested. The code in Listing 8-3 uses the DATEADD function to calculate the retirement vesting date for the top 5 employees based on HireDate.

Listing 8-3: Code to demo the DATEADD function

```
USE AdventureWorks2017;
GO
SELECT TOP 5 LoginID
           , HireDate
           , DATEADD(year,5,HireDate) as DateOfVesting
FROM HumanResources.Employee
ORDER BY HireDate;
```

You can see that I called the DATEADD functions with the parameter values: “year”, “5”, and “HireDate”. The “year” parameter identifies the datepart unit that will be used when the

DATEADD function calculates the new date. The parameter value of "5" indicates the number of datepart units, in this case years, that needs to be added to the "HireDate" to calculate the DateofVesting date. You use positive numbers to calculate dates after the date parameter and negative numbers to calculate dates before the date parameter. When the code in Listing 8-3 is run, the results in Report 8-3 are produced.

Report 8-3: Output when running the code in Listing 8-3

```
LoginID                    HireDate     DateOfVesting
-------------------------  ----------   -------------
Adventure-works\guy1       2006-06-30   2011-06-30
Adventure-works\kevin0     2007-01-26   2012-01-26
Adventure-works\roberto    2007-11-11   2012-11-11
Adventure-works\rob0       2007-12-05   2012-12-05
Adventure-works\thierry0   2007-12-11   2012-12-11
```

String Function Examples

Another subcategory of scalar functions is string functions. String functions are useful for parsing, concatenating and manipulating string values. There are lots of different built-in string functions available in SQL Server. Some functions are used to take apart strings, others put string values together, while others are used to probe into string values. A complete list of all the different string functions can be found in the this Microsoft documentation. To get a feel for how to use string functions to accomplish some TSQL programming situations, let me show you an example of how to parse apart a string into its pieces.

Parsing apart a String Value

Suppose you have a string that contains a name-value pair, where the name and the value are separate by a comma. This example uses the string "FirstName, Greg". Let me show you how to use a few different string functions to parse this string into two different local variables (@Name, and @Value) using the code in Listing 8-4.

Listing 8-4: Code to parse a string

```
DECLARE @String char (15) = 'FirstName, Greg';
DECLARE @Name char(9);
DECLARE @Value varchar(4);
SELECT @Name = SUBSTRING(@String,1,CHARINDEX(', ',@String)-1),
       @Value = SUBSTRING(@String,CHARINDEX(', ',@String)+2,LEN(@String));
SELECT @Name AS [Name], @Value AS [Value];
```

The code in Listing 8-4 uses the following string functions: SUBSTRING,CHARINDEX, and LEN.

The SUBSTRING function is used to parse the @String variable into the name (@Name), and value (@Value) pieces. The first reference to the SUBSTRING function parses the @String value into the @Name piece. I know that the name piece starts in character 1 of the @String variable and ends just before the comma. To identify the location of the comma, I use the CHARINDEX function. This function is used to identify the offset location within the @String variable where the string "," is found. More information about the CHARINDEX function: https://docs.microsoft.com/en-us/sql/t-sql/functions/charindex-transact-sql?view=sql-server-ver15.

The second reference to the SUBSTRING function is used to populate the @Value variable from the @String variable. The CHARINDEX function in this second reference is used to identify the starting location of the value portion of the name/value string. Just like before, I use the comma to identify this starting offset location, but, in addition, I need to identify the ending location for the value portion. To find the ending location, I just need to know the length of @String variable value. To do that I use the LEN function.

More information about the LEN function can be found in the Microsoft Documentation https://docs.microsoft.com/en-us/sql/t-sql/functions/len-transact-sql?view=sql-server-ver15.

When I run the code in Listing 8-4, I get the output shown in Report 8-4.

Report 8-4: Output from executing code in Listing 8-4

```
Name       Value
---------  -----
FirstName  Greg
```

There are lots of different string functions that you can use to parse, build and manipulate a string. I suggest your review Microsoft Documentation identified in this section to get a better understanding of how string functions can be used to support your TSQL application.

Conversion Function Example

There are times when you might need to convert an expression into an appropriate date type so further operations can be performed on it. SQL Server has the CAST and CONVERT functions to perform conversions. To better understand how each of these functions can be used, let me show you an example that uses both of these conversion functions. But first, let's review the syntax for the CAST and CONVERT functions found in this Microsoft documentation in Figure 8-3.

```
-- CAST Syntax:
CAST ( expression AS data_type [ ( length ) ] )

-- CONVERT Syntax:
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

expression
Any valid expression.

data_type
The target data type. This includes **xml**, **bigint**, and **sql_variant**. Alias data types cannot be used.

length
An optional integer that specifies the length of the target data type, for data types that allow a user specified length. The default value is 30.

style
An integer expression that specifies how the CONVERT function will translate *expression*. For a style value of NULL, NULL is returned. *data_type* determines the range.

Figure 8-3: Syntax for CAST and CONVERT functions

Note that the two functions are similar, but the CONVERT function also supports an additional style parameter. The style parameter is to help identify if any translations might need to be performed when expressions are converted, like how a floating point number should be represented, or does a date contain the century value or not.

For my first example, I will run the code in Listing 8-5.

Listing 8-5: Using CAST and CONVERT function

```
USE AdventureWorks2017;
GO
DECLARE @String varchar(3) = '123';
DECLARE @CurrentDateTime DATETIME = SYSDATETIME();
SELECT CAST (@String as INT) + 1 AS ConvertedString
      ,@CurrentDateTime as CurrentDateTime
      ,CONVERT (varchar,@CurrentDateTime, 103) AS OrderDate
      ,CONVERT(TIME,@CurrentDateTime,114) AS Time24HourFormat;
```

If you review the code in Listing 8-5, you will see that it uses the CAST function to convert the string value “123” into an integer value so integer math could be performed. Additionally, the code passes the system time in variable @CurrentDateTime to the CONVERT function twice. The first CONVERT function call converts the @CurrentDateTime value into a VARCHAR data

type value. The style format used (103) tell the function to make sure that the @CurrentDateTime value is outputted in dd/mm/yyyy format. The second CONVERT function call is used to covert the @CurrentDateTime value into a TIME data type, where the style parameter tells SQL Server to display the time value using a 24-hour format. When I run the code in Listing 5, I get output results in Report 8-5.

Report 8-5: Output when running code in Listing 8-5

```
ConvertedString   CurrentDateTime           OrderDate   Time24HourFormat
---------------   -----------------------   ----------  ----------------
124               2020-04-23 08:56:31.067   23/04/2020  08:56:31.0666667
```

I'd suggest you review the documentation to see what other style formats can be used to produce date values in other formats. The CAST and CONVERT functions are helpful in formatting your data however you need it formatted and displayed.

Using Scalar Function to Support Application Requirements

Scalar functions provide lots of different methods for managing and manipulating your SQL Server data. In this chapter, I was only able to demo a few of the commonly used scalar functions. I would suggest you spend some additional research time exploring all the available built-in function and how to use them. Understanding what built-in functions are available out of the box will go a long way to helping you better understand built-in functions and how they can support your application coding requirements.

Chapter 9

Summarizing Data Using a Simple GROUP BY Clause

In Chapter 6, I showed you how to sort your data using the ORDER BY clause. This allowed you to organize your detailed records in sort order based on single, or multiple columns. Detailed data is great if you want to see data in specific records, but sometimes you need to roll up the detailed data into summarized values. Summarizing your data can be accomplished using the GROUP BY clause.

There are two types of GROUP BY clauses. One that is known as the simple GROUP BY clause and another that provides a little more than simple summarization that is called the general GROUP BY clause. The main difference between these two types is the simple GROUP BY contains just the GROUP BY clause, whereas the general GROUP BY clause contains other operators like ROLLUP and CUBE.

In this article, I will cover how to group data using the simple GROUP BY clause. In a follow-up article, I will cover the more complex general GROUP BY clause.

Simple GROUP BY clause

Using the simple GROUP BY clause allows you to aggregate your data based on a single column, multiple columns or expressions. Only one summarized row will be returned for each unique value based on the columns and/or expressions specified in the GROUP BY clause. When SQL Server processes a GROUP BY clause, it groups the detailed records by the unique column or expression values. It then summarizes each set based on the aggregation functions included in the select list.

To better grasp how to use the GROUP BY, let's assume you have a table that contains detailed sales information for different stores, and you want to summarize total sales amount by store. You can use the GROUP BY clause to aggregate the total sales amount by each store. In this example, the unique column you would group on would be store name, and the column to be aggregated would be the sales amount. Your results would show one row for each unique store name, and the row for each store would contain the sum of the sales amounts for that store.

SQL Server has some limitation on what columns can be included in the SELECT list of a GROUP BY query. Each column specified in the SELECT list of a GROUP BY query needs to fall into one of the following categories:

- A column specified in the GROUP BY clause
- An expression specified in the GROUP BY clause
- The value returned from an aggregate function

If a column doesn't fall into one of these categories, you will get an error when trying to run your GROUP BY query. Note that a column or expression contained in the GROUP BY clause is not required to be in the select list.

Let me go through a couple of examples to help demonstrate how to obtain summarized values using the simple GROUP BY clause.

Sample Data for Exploring the Simple GROUP BY Clause

In order to demonstrate how to use a simple GROUP BY clause, I need to build some sample data. I am providing a script to create my sample data so you can run the sample code provided in this article. Use the script in Listing 9-1 to build and populate the sample tables.

Listing 9-1: Script to create Sample Data

```
USE tempdb;
GO
SET NOCOUNT ON;
-- Create Sales Table
CREATE TABLE dbo.SalesTransaction
   (Id INT IDENTITY PRIMARY KEY
   ,CustomerName VARCHAR(65)
   ,TotalSalesAmount money
   ,SalesTypeDesc VARCHAR(200)
   ,SalesDateTime DATETIME
   ,StoreName VARCHAR(100));

-- Add data to Sales Table
INSERT INTO dbo.SalesTransaction
      VALUES ('John Smith', 124.23,'Software','09/22/2011 11:51:12 AM','The
Software Outlet');
INSERT INTO dbo.SalesTransaction
      VALUES ('Jack Thomas', 29.56,'Computer Supplies','09/23/2011 10:21:49
AM','The Software Outlet');
INSERT INTO dbo.SalesTransaction
      VALUES ('Sue Hunter', 89.45,'Computer Supplies','09/23/2011 2:51:56
AM','The Software Outlet');
INSERT INTO dbo.SalesTransaction
      VALUES ('Karla Johnson', 759.12,'Software','09/23/2011 2:54:37
PM','The Software Outlet');
      INSERT INTO dbo.SalesTransaction
      VALUES ('Gary Clark', 81.51,'Software','09/22/2011 11:08:52
AM','Discount Software');
INSERT INTO dbo.SalesTransaction
      VALUES ('Scott Crochet', 12345.78,'Computer Supplies','09/23/2011
3:12:37 PM','Discount Software');
```

```
INSERT INTO dbo.SalesTransaction
      VALUES ('Sheri Holtz', 12.34,'Software','09/23/2011 10:51:42
AM','Discount Software');
INSERT INTO dbo.SalesTransaction
      VALUES ('Mary Lee', 101.34,'Software','09/23/2011 09:37:19
AM','Discount Software');
      INSERT INTO dbo.SalesTransaction
      VALUES ('Sally Davisson', 871.12,'Software','09/22/2011 05:21:28
PM','Discount Software');
INSERT INTO dbo.SalesTransaction
      VALUES ('Rod Kaplan', 2345.19,'Computer Supplies','09/23/2011 5:01:11
PM','Discount Software');
INSERT INTO dbo.SalesTransaction
      VALUES ('Sandy Roberts', 76.38,'Books','09/23/2011 4:51:57
PM','Computer Books and Software');
INSERT INTO dbo.SalesTransaction
      VALUES ('Marc Trotter', 562.94,'Software','09/23/2011 6:51:43
PM','Computer Books and Software');
```

If you look through the script in Listing 9-1, you will find I created the *dbo.SalesTransaction* table. I then inserted several records into this table. I will use this table to demonstrate how to use a simple GROUP BY clause to aggregate data.

Grouping by a Single Column

Using the sample table created using Listing 9-1, this first example will use the GROUP BY clause to summarize data based on a single column. My example in Listing 9-2 summarizes my sample data based on the *StoreName* column.

Listing 9-2: GROUP BY based on Single columns

```
USE tempdb;
GO
SELECT StoreName
     ,SUM(TotalSalesAmount) AS StoreSalesAmount
FROM dbo.SalesTransaction
GROUP BY StoreName;
```

When the code in Listing 9-2 is executed against my sample table, the following aggregated rows in Report 9-1 are returned.

Report 9-1: Summarizing sample data based on a single column

```
StoreName                                  StoreSalesAmount
------------------------------------------ ----------------
Computer Books and Software                639.32
Discount Software                          15757.28
The Software Outlet                        1002.36
```

If you review the output in Report 9-1, you can see that only one aggregated row is returned for each unique value of *StoreName*. The *StoreSalesAmount* on each record is calculated by summing up the *TotalSalesAmount* column for each store's Sales records using the SUM function.

Grouping by Multiple Columns

There are times when you need to develop a report in which your data needs to be grouped by multiple columns. In order to accomplish this, all you need to do is add additional columns to the GROUP BY clause. When multiple columns are specified in the GROUP BY clause SQL Server aggregates the detailed rows based on each unique combination of values from the

columns in the GROUP BY clause. In Listing 9-3, I have expanded the query in Listing 9-2 by adding a second column to the GROUP BY clause.

Listing 9-3: GROUP BY based on single columns

```
USE tempdb;
GO
SELECT StoreName, SalesTypeDesc
     ,SUM(TotalSalesAmount) AS StoreSalesAmount
FROM dbo.SalesTransaction
GROUP BY StoreName, SalesTypeDesc;
```

When I run the code in Listing 9-3 against my sample data, I get the results in Report 9-2.

Report 9-2: Output from Running Listing 9-3

```
StoreName                   SalesTypeDesc      StoreSalesAmount
--------------------------- ------------------ ----------------
Computer Books and Software  Books              76.38
Discount Software            Computer Supplies  14690.97
The Software Outlet          Computer Supplies  119.01
Computer Books and Software  Software           562.94
Discount Software            Software           1066.31
The Software Outlet          Software           883.35
```

In Report 9-2, you can see that *StoreSalesAmount* is now summarized at the *StoreName* and *SalesTypeDesc* level. Also, note that the aggregated rows returned are not in sorted order based on the columns in the GROUP BY clause. If I wanted the summarized data to appear in *StoreName* order, then I would have needed to include an ORDER BY clause on the SELECT statement. I will leave it up to you to add the ORDER BY to the code in Listing 9-3 to return the summarized data in *StoreName* order.

Using an Expression in the GROUP BY Clause

There are times that you may want to group your data by something other than a specific column or set of columns. For example, you might want to summarize your data based on the first few characters of some VARCHAR column, or maybe just the date, or month of a DATETIME column. SQL Server allows you to specify expressions in the GROUP BY clause to accomplish this. An expression could be any valid expression that is based on a column in the detailed recordset that is being aggregated. To demonstrate how to use an expression in the GROUP BY clause, look at the code in Listing 9-4.

Listing 9-4: GROUP BY based on Single columns

```
USE tempdb;
GO
SELECT CONVERT(CHAR(10),SalesDateTime,101) AS SalesDate
     ,SUM(TotalSalesAmount) AS TotalSalesAmount
FROM dbo.SalesTransaction
GROUP BY CONVERT(CHAR(10),SalesDateTime,101);
```

In Listing 9-4, the SELECT statement is grouping the data based on an expression, in this case, a CONVERT function. If you use an expression in the SELECT list, the same exact expression must be used in the GROUP BY clause. The CONVERT function is parsing the *SalesDateTime* column and returning only the date portion of this column. Using the CONVERT function in the GROUP BY clause allows me to summarize the *Sales* data based on the actual dates of the different *Sales* records. By doing this, I was able to summarize my sample to get the *TotalSalesAmount* by date for all stores, as shown in Report 9-3.

Report 9-3: Output when summarizing data based on expression

```
SalesDate  TotalSalesAmount
---------- ----------------
09/22/2011 1076.86
09/23/2011 16322.10
```

Using expressions allows you to programmatically identify which parts of your detailed data will be used to aggregate your data.

HAVING Clause

If you are aggregating data with the GROUP BY clause, you might want not to return all the aggregated values. Instead, you may want only to return a subset of the aggregated values. The HAVING clause can be used to selectively identify the aggregated values you want to return from the GROUP BY summarization.

Normally when we SELECT data, we use the WHERE clause to restrict the rows that are returned. The only problem with that is that the WHERE clause operates on row values, and not aggregated values. Therefore, the WHERE clause is unable to use aggregated values created by the GROUP BY clause. However, adding a HAVING clause after your GROUP BY clause provides you with a way to specify conditions to identify the specific summarized values that you want to be returned. To better understand this, let me provide you with a couple of examples.

One of the common things the HAVING clause might be used for when looking at store sales data is to determine the stores that are not meeting a specific sales quota. If you wanted to find all the stores that didn't meet a minimum sales amount you could do that with the code in Listing 9-5.

Listing 9-5: Restricting result set by using a HAVING clause

```
USE tempdb;
GO
SELECT StoreName
     ,SUM(TotalSalesAmount) AS StoreSalesAmount
FROM dbo.SalesTransaction
GROUP BY StoreName
HAVING SUM(TotalSalesAmount) < 1000.00;
```

In Listing 9-5, I restricted the result set to those stores that had an aggregated *TotalSalesAmount* of less than 1000.00. In my trivial example here, you will find the *StoreName* of "Computer Books and Software" is the only store that didn't meet the $1000.00 sales quota amount.

The HAVING clause can be used on columns that are also not aggregated, but normally filtering based on colunn values is done in the WHERE clause. If you want to restrict the rows returned based on a specific value of any one of the columns used in the GROUP BY clause you can also do that, and Listing 9-6 demonstrates this.

Listing 9-6: Restricting result set based on a GROUP BY column

```
USE tempdb;
GO
SELECT StoreName
     ,SUM(TotalSalesAmount) AS StoreSalesAmount
FROM dbo.SalesTransaction
GROUP BY StoreName
HAVING StoreName LIKE '%Outlet%'
    OR StoreName LIKE '%Books%';
```

In Listing 9-6, I only wanted to see summarized data for stores that have either "Outlet" or "Books" in their store name This example also demonstrates that you can have multiple

conditions in the HAVING clause. Another way to think of the difference between WHERE and HAVING is that the WHERE clause filters out data rows before the data is aggregated, and the HAVING clause filters out aggregated rows after the GROUP BY is applied.

Summarizing Data with the Simple GROUP BY Clause

In this chapter, I showed you how to use the simple GROUP BY clause to summarize your data. I discussed how to use a single column, multiple columns, as well as expressions in the GROUP BY clause to summarize detailed data. By using what I have demonstrated, you should now be able to build a simple GROUP BY clause to summarize your data, and optionally filter the summarized data using HAVING.

In the next chapter, I will expand my discussion of the GROUP BY clause. In this follow-up chapter, I will show you how to use the CUBE and ROLLUP operators to produce additional summarized data, such as sub-total and grand total values.

Chapter 10

Using the ROLLUP, CUBE and GROUPING SET operator in a GROUP BY Clause

In this chapter, I will expand on my discussion of the GROUP BY clause, which I started in Chapter 7, by exploring the ROLLUP, CUBE and GROUPING SETS operators. These additional GROUP BY operators make it is easy to have SQL Server create subtotals, grand totals, a superset of subtotals, as well as multiple aggregate groupings in a single SELECT statement. I will explain the differences between each of these different grouping operators and provide you with examples to demonstrate how they work.

When are the ROLLUP, CUBE, and GROUPING SETS operators useful?

The ROLLUP, CUBE and GROUPING SETS operators produce multiple different aggregate groupings. These operators are useful when you want to aggregate your data in multiple different ways. If you need to aggregate your data on several different column groupings, then you could use one of these three operators along with the GROUP BY clause to create those additional aggregation groupings.

Suppose you wanted to keep track of where you spent your money, not only at the individual transaction level but also at different levels of summarization, for example, based on different categories. With a standard GROUP BY clause, you can get the total the amount you spent at Target and Amazon.com on clothes. But what if you also wanted a single SELECT statement

to return the total you spent on clothes regardless of where you bought them? Or maybe you want to summarize your expenses based on additional categories, like food or utilities? Plus, what if you also want that single SELECT statement to return the grand total of all of your expenses. To accomplish these additional aggregation groupings, you can use the ROLLUP, CUBE or GROUPING SETS operators.

In the sections below, I will explain each of these operators in detail but first, let me build some sample data.

Sample Data

In order to demonstrate how to use these different GROUP BY operators, I will create a table that contains some sample data that I can summarize. To provide a little context that we all might be familiar with, I will create a table that will contain check registry information. My table will be named *CheckRegistry,* and it will contain the basic columns needed to record the information you might have filled out when writing a check. I will populate my *CheckRegistry* with multiple rows of data, each of which represents a check written to a business over four months.

If you want to follow along and run the examples I discussed later in the article, you can create my sample *CheckRegistry* table by running the code in Listing 10-1.

Listing 10-1: Create the CheckRegistry Table

```
SET NOCOUNT ON;
USE tempdb;
GO
CREATE TABLE CheckRegistry (
      CheckNumber smallint,
      PayTo varchar(20),
      Amount money,
      CheckFor varchar(20),
      CheckDate date);
```

```
INSERT INTO CheckRegistry VALUES
    (1000,'Seven Eleven',12.57,'Food','2011-07-12'),
    (1001,'Costco',128.57,'Clothes','2011-07-15'),
    (1002,'Costco',21.87,'Clothes','2011-07-18'),
    (1003,'AT&T',69.23,'Utilities','2011-07-25'),
    (1004,'Comcast',45.95,'Utilities','2011-07-25'),
    (1005,'Northwest Power',69.18,'Utilities','2011-07-25'),
    (1006,'StockMarket',59.25,'Food','2011-07-25'),
    (1007,'Safeway',120.21,'Food','2011-07-28'),
    (1008,'Albertsons',9.15,'Food','2011-010-02'),
    (1009,'Amazon',158.34,'Clothes','2011-010-05'),
    (1010,'Target',89.21,'Clothes','2011-010-06'),
    (1011,'AT&T',69.23,'Utilities','2011-010-25'),
    (1012,'Comcast',45.95,'Utilities','2011-010-25'),
    (1013,'Nordstrums',259.12,'Clothes','2011-010-27'),
    (1014,'AT&T',69.23,'Utilities','2011-09-25'),
    (1015,'Comcast',45.95,'Utilities','2011-09-25'),
    (1016,'Northwest Power',71.35,'Utilities','2011-09-25'),
    (1017,'Safeway',123.25,'Food','2011-09-25'),
    (1018,'Albertsons',65.11,'Food','2011-09-29'),
    (1019,'McDonalds',12.57,'Food','2011-09-29'),
    (1020,'AT&T',69.23,'Utilities','2011-10-25'),
    (1021,'Comcast',45.95,'Utilities','2011-10-25'),
    (1022,'Black Angus',159.23,'Food','2011-10-25'),
    (1023,'TicketMasters',59.87,'Entertainment','2011-10-30'),
    (1024,'WalMart',25.11,'Clothes','2011-10-31'),
    (1025,'Albertsons',158.50,'Food','2011-10-31');
```

Using the ROLLUP operator to Create Subtotals and a Grand Total

The ROLLUP operator can be used as part of the GROUP BY clause to provide subtotals and a grand total while your data is being aggregated. To understand how to use the ROLLUP operator, let me show you a few different examples. I'll start by showing you how to create a grand total of the amounts in our *CheckRegistry* table, by using the code in Listing 10-2.

```
USE tempdb;
GO
SELECT CheckFor
     , SUM (Amount) As Total
FROM CheckRegistry
GROUP BY ROLLUP (CheckFor);
```

Listing 10-2: Generating a Grand Total Row using ROLLUP on the *CheckFor* column

When the code in Listing 10-2 is executed, the output in Report 10-1 is returned.

Report 10-1: Output produced when running code in Listing 10-2

CheckFor	Total
Clothes	682.22
Entertainment	59.87
Food	719.84
Utilities	601.25
NULL	2063.18

If you look at the output in Report 10-1, you can see that SQL Server summarized values in the *Amount* column for each of the different *CheckFor* values. Additionally, a final row was displayed that contains the grand total of the *Amount* values for all the checks written. The grand total row can be identified by the NULL in the *CheckFor* column.

I've never liked the NULL being returned for the grand total value when you use the ROLLUP operator. I would prefer that it said "GRAND TOTAL". To accomplish this, you can modify the code from Listing 10-2 slightly, as shown in Listing 10-3. In Listing 10-3, I added a COALESCE function on the *CheckFor* column. The COALESCE function allowed me to replace the NULL that was displayed in Report 10-1 with "GRAND TOTAL". You can run this code for yourself to verify that "GRAND TOTAL" is replacing the NULL.

Listing 10-3: Replacing "NULL" reference with "GRAND TOTAL"

```
USE tempdb;
GO
SELECT COALESCE (CheckFor,'GRAND TOTAL') As CheckFor
     , SUM (Amount) AS Total
FROM CheckRegistry
GROUP BY ROLLUP (CheckFor);
```

If you include more than one column in the ROLLUP specifications, it will create subtotals for each column in the ROLLUP specification along with the grand total. The creation of the subtotal amounts will be done based on aggregating the columns in the ROLLUP specifications from right to left. In other words, if you have a ROLLUP clause that has only two columns, there will be two different sets of subtotals, one for each column in the ROLLUP clause, where the more granular subtotals will be based on the rightmost column. If you have three columns identified in the ROLLUP specification, then SQL Server will create the first set of subtotal amounts based on the rightmost column (third column identified in the ROLLUP specification). Then the second set of subtotal will be based on the second column identified in the ROLLUP specification. To better understand the aggregation order from right to left, let me go through a couple of examples.

For the first example, my goal is to summarize my check registry information by month. I want to know how much I spent every month on clothes, entertainment, food, etc. In addition, I also want a rolled up amount that shows me the total amount I spent each month. I can do this by identifying my two different rollup criteria in the ROLLUP specification. My first criterion will identify what I wrote my checks for, which is specified by using the *CheckFor* column. The second criterion will identify the month the check was written, which can be calculated using the MONTH function against the *CheckDate* column.

To get my data rolled up in the correct order, I will need to specify each of my criteria in the correct order. Remember that data is aggregated using the columns in the ROLLUP specifications in right to left order. Or another way to say is that is the column that requires

the most granular rollup needs to be specified last in the ROLLUP criteria. In my requirements above, I wanted the *CheckFor* column to be aggregated first. To meet my summarization requirements, I can use the code in Listing 10-4.

Listing 10-4: Generating Subtotals and Grand Total

```
USE tempdb;
GO
SELECT MONTH(CheckDate) AS CheckMonth
     , CheckFor
     , SUM (Amount) AS Total
FROM CheckRegistry
GROUP BY ROLLUP (MONTH(CheckDate),CheckFor);
```

When I run the code in Listing 10-4, I get the results in Report 10-2:

Report 10-2: Output from running Listing 10-4

CheckMonth	CheckFor	Total
7	Clothes	150.44
7	Food	192.03
7	Utilities	184.36
7	NULL	526.83

8	Clothes	506.67
8	Food	9.15
8	Utilities	115.18
8	NULL	631.00
9	Food	200.93
9	Utilities	186.53
9	NULL	387.46
10	Clothes	25.11
10	Entertainment	59.87
10	Food	317.73
10	Utilities	115.18

10	NULL	517.89
NULL	NULL	2063.18

In the output that is displayed in Report 10-2, you can see that for each *CheckMonth* there is an aggregated amount displayed for each of the different *CheckFor* values, which is found in the *Total* column. These values were created by specifying *CheckFor* as the rightmost criterion in the ROLLUP specification. Also included in the results is a row for every *CheckMonth* value that has a value of NULL in the *CheckFor* column. This monthly value is the total amount of checks written for that month and was created by specifying *MONTH(CheckDate)* as the leftmost column in the ROLLUP criteria. If you look at the last row in the results, you can see a row that has NULL in both the *CheckMonth* and *CheckFor* columns. This is the row that contains the grand total for the *Amount* column in my *CheckRegistry* table.

If you specify more than two columns in the ROLLUP specification SQL Server will create subtotals for each of the additional columns. Listing 10-5 is an example that creates two different subtotal amounts by specifying three columns in the ROLLUP specification.

Listing 10-5: Creating Multiple Subtotal rows

```
USE tempdb;
GO
SELECT MONTH(CheckDate) AS CheckMonth
     , CheckFor
     , PayTo
     , SUM (Amount) AS Total
FROM CheckRegistry
GROUP BY ROLLUP (MONTH(CheckDate),CheckFor,PayTo);
```

I'll leave it up to you to run this code and review the output. When you do that, you should see subtotals for both the *Checkfor* and *CheckMonth* columns.

Using the CUBE Operator to create a Superset of Aggregated Values

The name CUBE comes from the concept of aggregating data along N dimensions of data, which when aggregating data across three dimensions just happens to be represented by a CUBE. If you would like to find out more about the research Microsoft did and the concepts behind the CUBE operator, there is a great research paper you can read that can be found here: http://research.microsoft.com/apps/pubs/default.aspx?id=69578.

The CUBE operator, just like the ROLLUP operator, creates subtotals and grand totals, but it also creates aggregates across all the different columns identified in the CUBE specifications or what I will call a superset of aggregated values. Therefore, the number of possible summarized values could be substantial when you consider that the superset would contain summarized values for all the permutations of the columns involved in the CUBE specification. To demonstrate how this works, let me go through a couple examples, starting with the code in Listing 10-6.

Listing 10-6: Creating SubTotals and GrandTotal using Cube Specification

```
USE tempdb;
GO
SELECT CheckFor
     , SUM (Amount) As Total
FROM CheckRegistry
GROUP BY CUBE(CheckFor);
```

If you run the code in Listing 10-6, you will see that the results produced will contain aggregated values for the Amount column for each of the *CheckFor* values. Additionally, the results will contain a grand total amount for all the *Amount* values. The results are the same

as the results produced by Listing 10-2 using ROLLUP and shown in Report 10-1. This is because there is only a single value referenced in the CUBE specification, so SQL Server only had to aggregate values based on a single column.

To better demonstrate how the CUBE operator creates aggregates for every permutation of the columns referenced in the CUBE specifications, let me run the code in Listing 10-7.

Listing 10-7: Using two columns in the CUBE Specification

```
USE tempdb;
GO
SELECT MONTH(CheckDate) AS CheckMonth
     , CheckFor
     , SUM (Amount) AS Total
FROM CheckRegistry
GROUP BY CUBE (MONTH(CheckDate),CheckFor);
```

This code will create aggregates for all the permutations of *MONTH(CheckDate)* and *CheckFor* values. That means this code will produce summarized values for the following permutations:

- CheckFor
- CheckFor and MONTH(CheckDate)
- MONTH(CheckDate)
- Grand total

If you review the result set in Report 10-3, you will see these aggregations.

Report 10-3: Results when using two columns in CUBE specifications

CheckMonth	CheckFor	Total
7	Clothes	150.44
8	Clothes	506.67
10	Clothes	25.11
NULL	Clothes	682.22
10	Entertainment	59.87
NULL	Entertainment	59.87
7	Food	192.03
8	Food	9.15
9	Food	200.93
10	Food	317.73

NULL	Food	719.84
7	Utilities	184.36
8	Utilities	115.18
9	Utilities	186.53
10	Utilities	115.18
NULL	Utilities	601.25
NULL	NULL	2063.18
7	NULL	526.83
8	NULL	631.00
9	NULL	387.46
10	NULL	517.89

In the results in Report 10-3, you can see that by using the CUBE specification, SQL Server aggregated each *CheckFor* value by month and then calculated a subtotal for each *CheckFor* value. Additionally, SQL Server calculating the grand total (the row with NULL in both the *CheckMonth* and *CheckFor* columns). The last few rows in the report show monthly subtotal aggregations for each *CheckMonth*.

If you start adding additional columns to the CUBE specification, the number of aggregations that SQL Server will create will grow substantially. This is because every combination of the columns in the CUBE specification will be aggregated. Run the code in Listing 10-8 to see how many summarized values will be produced when using a CUBE specification with three columns.

Listing 10-8: CUBE specification using three columns

```
USE tempdb;
GO
SELECT MONTH(CheckDate) AS CheckMonth
     , CheckFor
     , PayTo
     , SUM (Amount) AS Total
FROM CheckRegistry
GROUP BY CUBE (MONTH(CheckDate),CheckFor, PayTo);
```

Using the CUBE specification is very useful when you want to summarize data in a Data Warehouse situation. The CUBE specification provides an easy mechanism to create subtotals across many different combinations of dimensions.

Creating Multiple Aggregated Groupings using the GROUPING SETS Operator

Using the GROUPING SETS operator with a GROUP BY clause allows you to create a recordset that will aggregate your data multiple different ways. The GROUPING SETS specification allows you an alternative to writing a UNION ALL query where each SELECT statement is grouping data on a different column. The number of different aggregates is based on the number of columns or sets of columns provided in the GROUPING SETS specification. Let me demonstrate this by first showing you a GROUPING SETS query (Listing 10-9) and then the equivalent UNION ALL query (Listing 10-10).

Listing 10-9: GROUPING SETS query

```
USE tempdb;
GO
SELECT MONTH(CheckDate) AS CheckMonth
     , CheckFor
     , SUM (Amount) AS Total
FROM CheckRegistry
GROUP BY GROUPING SETS (MONTH(CheckDate),CheckFor);
```

When I run my GROUPING SETS query in Listing 10-9, I get the result set shown in Report 10-4:

Report 10-4: Result set produced with running code in Listing 10-9

CheckMonth	CheckFor	Total
NULL	Clothes	682.22
NULL	Entertainment	59.87
NULL	Food	719.84
NULL	Utilities	601.25
7	NULL	526.83
8	NULL	631.00
9	NULL	387.46
10	NULL	517.89

By looking at Report 10-4, you can see that the first 4 rows of output have aggregates created by grouping my sample data based on the *CheckFor* column. Then the last 4 rows are aggregated based on the calculated *CheckMonth* column. My GROUPING SETS query in Listing 10-9 is equivalent to running the UNION ALL query in Listing 10-10.

Listing 10-10: UNION ALL query equivalent to Listing 10-9

```
USE tempdb;
GO
SELECT NULL AS CheckMonth
     , CheckFor
     , SUM (Amount) AS Total
FROM CheckRegistry
GROUP BY CheckFor
UNION ALL
SELECT MONTH(CheckDate) AS CheckMonth
     , NULL as CheckFor
     , SUM (Amount) AS Total
FROM CheckRegistry
GROUP BY MONTH(CheckDate);
```

The GROUPING SETS operator can also be used to create similar result sets produced by ROLLUP and CUBE operators. To learn more about this, refer to the books online topic "GROUPING SETS Equivalents".

You can use the GROUPING SETS operator to aggregate values based on more than two columns. You do this by putting the multiple columns you want to group by inside of parentheses. The code in Listing 10-11 demonstrates this:

Listing 10-11: Aggregating on multiple columns

```
USE tempdb;
GO
SELECT MONTH(CheckDate) AS CheckMonth
     , CheckFor
     , PayTo
     , SUM (Amount) AS Total
FROM CheckRegistry
GROUP BY GROUPING SETS ((MONTH(CheckDate),CheckFor), PayTo);
```

I am creating two different aggregated sets in the code in Listing 10-11. The first aggregated set is based on the *PayTo column*, and the second aggregated set is based on the combination of the month the check was written and the *CheckFor* column.

You can even create a grand total amount by introducing the empty set "()" representation into the GROUPING SET specifications. If you run the code in Listing 10-12, you will see that one extra row is created in addition the rows shown in Report 10-3. This additional grand total row will have a NULL value identified for both the CheckMonth and CheckFor columns.

Listing 10-12: Creating a Grand Total row using the empty set in the GROUPING SETS specification

```
USE tempdb;
GO
SELECT MONTH(CheckDate) AS CheckMonth
     , CheckFor
     , SUM (Amount) AS Total
FROM CheckRegistry
GROUP BY GROUPING SETS (MONTH(CheckDate),CheckFor,());
```

If you want to produce multiple aggregated groupings with minimal code, then the GROUPING SETS operator is a way to accomplish this.

More than One Way to Aggregate Data: Using ROLLUP, CUBE and GROUPING SETS

As in the old saying “There is more than one way to skin a cat”, there is also more than one way to aggregate data in T-SQL. In addition to using the simple GROUP BY clause that was introduced in Chapter 7, you can also use the ROLLUP, CUBE and GROUPING SETS specification to create subtotals, grand totals, and a superset of aggregated values. These additional GROUP BY operators are very useful in creating different aggregations to meet your application needs.

Chapter 11

Adding Records to a table using INSERT Statement

Most of the previous chapters have dealt with selecting data from SQL Server tables. Not all applications are limited to only retrieving data from a database. Your application might need to insert, update or delete data as well. In this chapter, I will discuss various ways to insert data into a table using an INSERT statement. Later chapters will explore updating and deleting SQL Server data.

The Basic INSERT statement

There are a number of formats the INSERT statement can take. When I refer to the basic INSERT statement, I'm referring a simple INSERT statement that includes a list of column names and values for each of those columns.

In order to demonstrate how to use a basic INSERT statement, we need to have a table into which we can insert data. For the purpose of this chapter, I'm going to create a very simple table name *Fruit.* The *Fruit* table will track the different varieties of fruit and the quantity of fruit boxes stored in a warehouse. My *Fruit* table will contain the following columns, *Id, Name, Color,* and *Quantity*. The *Id* column will be an integer value that I use to uniquely identify each type of fruit. The *Name* column is a *varchar* value that contains a common name to refer to the fruit. The *Color* column will distinguish the different fruit colors if the particular fruit has multiple colors. And lastly, the *Quantity* column will track the actual number of boxes that are stored in the warehouse. If you want to run each INSERT statement in this chapter, you will need to create my Fruit table by running the code in Listing 11- 1:

Listing 11- 1: Create Fruit Table

```
USE tempdb;
GO
CREATE TABLE Fruit (
   Id int NOT NULL,
   Name varchar(100) NOT NULL,
   Color varchar(100) NULL,
   Quantity int DEFAULT 1);
```

As you can see, I made the *Id* and *Name* columns required fields by specifying the attribute NOT NULL. I also specified a default value for the *Quantity* column. The different constraints on these columns will determine how my INSERT statement can look.

The basic syntax for most INSERT statements you will write uses the following format:

```
INSERT (column_list) VALUES (value_list);
```

Where:

- column_list contains a list of columns in the inserted row, which will have a have a specific data value supplied
- value_list contains a list of data values supplied for the columns identified in the column_list specification.

The column_list specification is only needed if the value_list doesn't include a column value for each column in your table. Let me go through a few examples to demonstrate how to use the "column_list" and "value_list" appropriately.

For my first example, I will run the INSERT statement shown in Listing 11-2:

Listing 11-2: INSERT statement with a column_list and value_list that contains every column in table *Fruit*

```
INSERT INTO Fruit (Id, Name, Color, Quantity)
   VALUES (1, 'Banana', 'Yellow', 1);
```

In this INSERT statement, I have provided a column_list and value_list that contains every column in the *Fruit* table. This statement will insert one row into my table for a fruit called "Banana".

Alternatively, if I am providing a value for every single column in the Fruit table, I can leave off the column_list specification, as the INSERT statement in Listing 11- 3 shows.

Listing 11- 3: INSERT statement without the column_list specification

```
INSERT INTO Fruit
   VALUES (2, 'Grapes', 'Red', 15);
```

The column list specification for an INSERT statement at a minimum will need to identify a value for every column in your table that requires a value. If your table definition provides any default values or allows null values for columns, then those columns do not need to have a value supplied. The example in Listing 11- 4 shows an INSERT statement where I do not provide a value for *Color* and *Quantity*.

Listing 11- 4: INSERT statement that doesn't include all table columns

```
INSERT INTO Fruit (Id, Name)
   VALUES (3, 'Apples');
```

I don't need to provide a value for *Color* because it is defined to allow NULL values. The *Quantity* column doesn't need to be included because there is a default constraint associated with this column.

Occasionally, you might want to insert more than one record into a table at a time. You can do that using multiple INSERT statements, or you can use the syntax in Listing 11- 5, which uses the syntax for inserting multiple rows with a single INSERT statement.

Listing 11- 5: Inserting multiple records into a table with a single insert statement

```
INSERT INTO Fruit(Id, Name, Color, Quantity)
   VALUES (4, 'Apples', 'Red', 10),
          (5, 'Peaches', 'Green', 7),
          (6, 'Pineapples','Yellow', 5);
```

In Listing 11- 5, I inserted 3 different rows into my *Fruit* table with a single INSERT statement. This was accomplished by providing three different VALUES specifications separated by commas. Each of these different value statements contains a different type of fruit and will create a new row for each value.

Inserting Data into a Table using a SELECT statement

There are times when you want to insert a large number of records into a table that are based upon another recordset returned from a SELECT statement. In this case, it would be very cumbersome to insert records one at a time using the INSERT with the values_list method as described in the prior section. Instead, you can use the output of a SELECT statement as input into an INSERT statement, as my example in Listing 11- 6 demonstrates.

Listing 11- 6: Inserting multiple records into a table using a SELECT statement

```
USE tempdb;
GO
CREATE TABLE Fruit_Import (
   Id int NOT NULL,
   Name varchar(100) NOT NULL,
   Color varchar(100) NULL,
```

```
   Quantity int DEFAULT 1);
INSERT INTO Fruit_Import
      VALUES ('7','Watermelon','Green',100),
             ('8','Mango','Yellow',17);
INSERT INTO Fruit (ID, Name, Color, Quantity)
      SELECT ID, Name, Color, Quantity FROM Fruit_Import;
```

In Listing 11- 6, I first created a table named *Fruit_Import*. This table well be use to stage up records to be imported into the *Fruit* table. I then place two new rows of fruit into this staging table. Lastly, I used a SELECT statement in conjunction with the INSERT statement to get the new fruit rows from the *Fruit_Import* table so they could be inserted into the *Fruit* table.

Inserting Data into a Table using a Stored Procedure

There are times when a single SELECT statement is not enough to identify the records you want to insert into a table. You might have some complex logic to generate a number of rows that need to be inserted. When this is the case, you can easily build a stored procedure to produce a recordset that can be used to insert data into a table. In Listing 11- 7, I have a stored procedure that generates some hybrid fruit by concatenating fruit names together. The output of the stored procedure is then used to insert records into my *Fruit* table.

Listing 11- 7: Using a stored procedure to insert records into a table

```
CREATE PROC HybridFruit
AS
   SELECT b.Id + 9, a.Name + b.name
   FROM Fruit a INNER JOIN Fruit b
   ON a.Id = 9 - b.Id;
GO
INSERT INTO Fruit (Id, Name) EXECUTE HybridFruit;
```

In this example, I first create my stored procedure *HybridFruit*. This stored procedure joins the fruit table to itself in a single SELECT statement to create my new hybrid fruit names. I then use the output of this stored procedure as input into my INSERT statement (the last statement in Listing 11- 7). I do this by using the EXECUTE option of the INSERT statement.

Using the OUTPUT Clause

When you are inserting records into a table, you can also output the inserted values. These inserted values than can be used by the calling application or subsequent TSQL code. The code in Listing 11- 8 shows how to output inserted values so the calling application can retrieve the inserted values.

Listing 11- 8: Using OUTPUT clause to return inserted values to calling application

```
INSERT INTO Fruit(Id, Name)
   OUTPUT INSERTED.*
   VALUES (18,'Pie Cherries');
```

This example inserts a single row into the INSERTED table using the OUTPUT clause. The ".*" notation following the word INSERTED tells SQL Server to output the value for every inserted column value, even those that are generated, like default values. When I run this code from a query window within SQL Server Management Studio, it returns the data in the INSERTED table in the results pane. The results I got when I ran the code in Listing 11- 8 is shown in Report 11- 1.

ID	Name	Color	Quantity
18	Pie Cherries	NULL	1

Report 11- 1: Results returned to client from OUTPUT clause

If you review the output shown in Report 11- 1, you can see there a value for every column in my *Fruit* table, except the *Color* column that allowed nulls. Note that a value for the *Quantity* column is returned, even though I didn't use this column in my original query. By using the OUTPUT clause, you can obtain values for columns that are computed based on column constraints. By using the OUTPUT clause, you can now obtain the values for an identity column if your table had one.

When an OUTPUT clause is used without an INTO statement as I did, you cannot have an INSERT trigger defined on the table into which the rows are inserted. If you want to have a trigger on your table, you need to use the INTO option associated with the OUTPUT clause. By using the INTO option of an OUTPUT clause, you can retrieve the INSERTED column values into a table or table variable. Having those column values in a table allows you to build logic in your application that can process through the INSERTED data. My code in Listing 11- 9 alters the table to make the *Id* column an identity column, which means it will have its value automatically generated by SQL Server during the INSERT. The code then demonstrates how to use the INTO option to return the identity value and fruit name for each row inserted into my *Fruit* table.

Listing 11- 9: Using OUTPUT clause to return inserted values to calling application

```
-- Alter table so Id column is now an identity
ALTER TABLE Fruit
  DROP Column Id;
ALTER TABLE Fruit
 ADD Id int identity;
-- Create table variable to hold output from insert
DECLARE @INSERTED as TABLE (
   Id int,
   Name varchar(100));
--INSERT with OUTPUT clause
INSERT INTO Fruit (Name, Color)
   OUTPUT INSERTED.Id,
          INSERTED.Name
   INTO @INSERTED
   VALUES ('Bing Cherries','Purple'),
          ('Oranges','Orange');
          -- view rows that where inserted
SELECT * FROM @INSERTED;
```

By looking at the code in Listing 11- 9, you can see I first dropped the original *Id* column and then added a new *Id* column as an identity column. At the time the new column is added with the identity property, values will automatically be generated for each existing row. I then create my table variable @INSERTED that will hold the column values of any rows I insert into my *Fruit* table. I then insert two new rows into my *Fruit* table. If you look at my OUTPUT clause on the INSERT statement, you will see two things. First, you will notice that I specifically identified the INSERTED column values I wanted to output. In my example, that would be *Id* and *Name*. The second thing I did was include the INTO option, which identified that I wanted the output column values to be inserted into my table variable named @INSERTED. From this example, you can see that you don't have to OUTPUT all of the INSERTED columns, but only those you need. In my example, I only inserted the values *Id* and *Name* into my table

variable. Lastly, I selected the data that was inserted into my table variable @INSERTED. You can see the data that was placed in the table variable by reviewing the output shown in Report 11- 2.

Report 11- 2: Results of running Listing 11- 9

ID	Name
19	Bing Cherries
20	Oranges

Being able to capture the identity column values can be useful in helping you when your database design requires the identity value from a row to be used in other columns within your database.

One thing worth noting is that there are times when an OUTPUT clause cannot be used. One of those times is when using the EXECUTE option of the INSERT statement. For additional information on cases where an OUTPUT clause cannot be used refer to the “OUTPUT clause” topic in SQL Server’s online documentation.

Populating Data in a Table

When you want to have your application populate data into a table, the INSERT should be your statement of choice. There are many different ways to use the INSERT statement to populate data in a table, as I demonstrated in this chapter. By no means can a short chapter like this cover all INSERT statement options. The options I showed you are those options that are

most commonly used. If you want to learn more about all the different insert options, you should refer to SQL Server's documentation.

Chapter 12

Changing Data with the UPDATE Statement

Unless you are working on a reporting-only application, you will probably need to update tables in your SQL Server database. To update rows in a table, you use the UPDATE statement. There are several different ways you can use the UPDATE statement. In this chapter, I will discuss how to find and update records in your database and discuss the pitfalls you might run into when using the UPDATE statement.

Basic Syntax of UPDATE Statement

There are a number of different formats that the UPDATE statement can use. I will show you the UPDATE statement syntax that is most commonly used. Listing 12-1 shows the basic syntax for the UPDATE statement:

Listing 12-1: The UPDATE syntax

```
UPDATE <object_name>
    SET <column_name> =  [ ,…n]
    [ <OUTPUT Clause> ]
    [ FROM <table_source> [ ,…n]
    [ WHERE  <search_condition>
```

Where:

<object_name> - is the table that will be updated.

<column_name> - is a column that will be updated

<value> - is the string or number that will be used to update the <column_name>

<table_source> - is a table, view or derived table that provide the <value> to be used to update a <column_name>

<search_condition> - defines the criteria a row must meet in order to be updated.

For the complete syntax of the UPDATE statement, refer to the online SQL Server documentation.

In order to demonstrate the use of the UPDATE statement, I will need a couple of different tables. The code in Listing 12-2 can be used to create the tables I will use in my examples.

Listing 12-2: Code to create tables used in examples

```
SET NOCOUNT ON;
USE tempdb;
go
CREATE TABLE Product (
      ID int identity not null,
      ProductName varchar(25) not null,
      Price decimal(6,2) not null
      );
INSERT INTO Product
      VALUES ('Widget',25.99),
      ('WingDng',18.87),
             ('DingDong',1.99),
             ('DoDad',87.34);
```

```
CREATE TABLE New_Prices (
      ID int not null,
      ProductName varchar(25) not null,
      Price decimal(6,2) not null
      );
INSERT INTO New_Prices
      VALUES (1,'Widget',26.99),
             (2,'WingDing',19.31),
             (3,'DingDong',2.99),
              (4,'Doo-Dads',97.21);
```

Updating a Single Column in a Single Row

You may find you need to update data in a table for many different reasons. When I created my *Product* table, I incorrectly typed the names for some of my products. One of my typos is where I typed "WIngDng" when it should have been "WingDing". To update this single column on a single row, I can run the UPDATE statement in Listing 12-3.

Listing 12-3: Update a single column

```
UPDATE Product
  SET ProductName = 'WingDing'
WHERE ProductName = 'WingDng';
```

In Listing 12-3, I used the <search_condition> of the UPDATE statement syntax to identify the criteria that a row should meet for it to be updated. In this case, I need to search for the value "WingDng" in the *ProductName* column. By checking if the value in the *ProductName* column was equal to my misspelled string, I was able to identify the single row that needed to be updated. The SET clause of the UPDATE statement was used to specify the actual column value that should be used in *ProductName* column, namely "WingDing".

The pitfall of using the UPDATE Statement

Care needs to be taken when you are writing UPDATE statements to make sure you don't incorrectly update more or fewer rows than you want. One of the common problems you might run across is forgetting to provide a <search_condition> or having a <search_condition> that identifies too many or too few rows than the number you want to update. In Listing 12-4, I have a <search_condition> that causes my update statement to update more rows than I want.

Listing 12-4: Updating too many records

```
SET NOCOUNT OFF;
UPDATE Product
   SET Price = 19.27
WHERE ProductName like 'W%';
```

If you run the code in Listing 12-4, you will find it updated two rows because of my <search criteria> when my original intent was only to update the single "WingDing" row. The "SET NOCOUNT OFF" statement at the beginning of Listing 12-4 suppresses the message that shows the number of rows affected by the UPDATE statement.

If you are writing UPDATE statements to resolve some data integrity problem with data in a table and you are not exactly sure which rows will be updated using a particular <search_condition>, you should first write a SELECT statement using the <search_condition> you think is correct. Then once your SELECT statement returns the appropriate rows to be updated, you can then turn your SELECT statement into an UPDATE statement. This practice provides you with a method to help make sure only the correct rows are updated.

Changing Multiple Columns Using a Single UPDATE Statement

A single UPDATE statement can update more than a single column. This can be done by referencing multiple column names in the SET clause, as shown in Listing 12-5.

Listing 12-5: Updating two columns with a single UPDATE statement

```
UPDATE Product
   SET ProductName = 'Do-Dads',
       Price = 81.58
WHERE ProductName = 'DoDad';
```

In Listing 12-5, I updated both the *ProductName* and *Price* column with a single UPDATE statement. In this example, those columns will only be updated in the rows that had a *ProductNam*e of "DoDad".

Updating Columns Based on Column Values from another Table

You don't always have to use literal values to update columns when using the UPDATE statement. You can update columns in one table using the values found in another table. In Listing 12-6, I show how to use the FROM clause to use column values from one table to update another.

Listing 12-6: Updating columns based on records from another table

```
UPDATE Product
   SET ProductName = N.ProductName,
       Price = N.Price
FROM Product P JOIN New_Prices N
     ON P.ID = N.ID;
```

The code in Listing 12-6, uses the FROM clause to match rows between the *Product* and *New_Prices* tables based on the *ID* column. For every matching row, I take the *ProductName* and *Price* column from the *New_Prices* table to update the corresponding columns in the *Product* table.

You do not have to update every matching row when you use the FROM clause. If you add a WHERE condition like I have done in Listing 12-7, you can identify which matching rows are to be updated.

Listing 12-7: Constraining which columns to update based on a WHERE condition

```
UPDATE Product
   SET Price = N.Price
FROM Product P JOIN New_Prices N
     ON P.ID = N.ID
WHERE P.ProductName = 'Doo-Dads';
```

In Listing 12-7, I only updated the *Price* column on those rows that had a *ProductName* of "Doo-Dads".

Using the .WRITE Clause with an UPDATE statement

If you have columns that are defined using large data types in your table, like varchar(max), ncharchar(max) or varbinary(max), then you can use the .WRITE clause to update these columns. The .WRITE clause was introduced with SQL Server 2005. This clause can be used to update a substring or to append data to end of a large data type column.

Listing 12-8 shows the syntax of the .WRITE clause:

Listing 12-8: The .WRITE clause syntax

```
.WRITE ( expression, @Offset , @Length )
```

Where:

expression - is the character string you want to insert in the large character data type column

@Offset – is the starting potion where the *expression* will be written

@Length – is the number of characters that will be replaced by the *expression*

To show how to use the .WRITE clause, let me go through a couple of examples. For my first example which can be found in Listing 12-9, I will update some text near the end of a column that contains a very large character string. For this example, I start by creating a new table that contains the large data type column that I want to update.

Listing 12-9: Updating the middle of a large character string

```
SET NOCOUNT ON;
USE tempdb;
GO
CREATE TABLE MyDemo (LargeColumn varchar(max));
INSERT INTO MyDemo (LargeColumn)
   VALUES (REPLICATE(CAST('A' AS varchar(max)),60000)
                  + 'etadpu ot erehw si ereH');
SELECT REVERSE(LargeColumn) 'Reverse of LargeColumn'
   FROM MyDemo;
UPDATE MyDemo
  SET LargeColumn.WRITE('nmuloc ym detadpu evah I',60000,18);
SELECT REVERSE(LargeColumn) 'Reverse of LargeColumn' FROM MyDemo;
```

When I run the code in Listing 12-9, I get the output in Report 12-1.

Report 12-1: Output from running Listing 12-9

Reverse of LargeColumn
`Here is where to updateAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…`
Reverse of LargeColumn
`Here I have updated my columnAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…`

If you review the code in Listing 12-9, you will see I created a table *MyDemo* and inserted a very large character string, of over 60,000 characters in length, into column *LargeColumn*. Once the record is inserted, I display the *LargeColumn* value in the reverse order so you can see the character values at the end of the string. I then use the .WRITE clause to update 18 characters starting at offset 60,000 with the value of “nmuloc ym detadpu evah I”. I then use another SELECT statement that uses the REVERSE function again to display the end of the text value for the updated column.

The .WRITE clause is also useful for appending information to a large data column. In Listing 12-10, I use the .WRITE clause to add a character string to the end of my large text column named *LargeColumn*

Listing 12-10: Appending to the end of a large string

```
UPDATE MyDemo
  SET LargeColumn.WRITE(' .txet siht dedneppa won evah I'
           ,LEN(LargeColumn),0);
SELECT REVERSE(LargeColumn) 'Reverse of LargeColumn'
    FROM MyDemo;
```

When I run the code in Listing 12-10, I get the output in Report 12-2.

Report 12-2: Output from running Listing 12-10

Reverse of LargeColumn
`I have now appended this text. Here I have updated my columnAAAAAAAAAAAAAAAAAAAAAAA…`

The code in Listing 12-10 uses the LEN function to identity the offset to start appending characters to my *LargeColumn* value and then uses a length value of zero to identify that no characters are going to be replaced. In reality, any number could have been used to represent the length parameter of the .WRITE clause because I am adding characters at the end of the existing column value.

Note that you can't use the .WRITE clause to update a large column when the column contains a null value. You must first update the null value to a non-null value, and then you can use the .WRITE clause to update the large data type column.

Using the OUTPUT Clause with an UPDATE statement

When using the UPDATE statement, SQL Server populates two different pseudo tables named INSERTED and DELETED. The INSERTED pseudo-table will contain all of the column values for the entire updated rows after the UPDATE statement has completed. Whereas the DELETED pseudo-table contains the old column values for the entire update row prior to the update statement being performed. You can think of the DELETED pseudo-table as containing the before images of updated rows, whereas the INSERTED pseudo-table contains the after images of updated rows. By using these two tables, you can get the before and after images of rows that were updated. If you use the OUTPUT clause on an UPDATE statement, you can expose the values from these pseudo-tables to your application. To see how to use these pseudo-tables to return data to your calling application, review the code in Listing 12-11.

Listing 12-11: Using the OUTPUT clause on an UPDATE statement

```
UPDATE Product
   SET Price = 2.11
   OUTPUT DELETED.*,INSERTED.*
WHERE ProductName = 'DingDong';
```

When I run the code in Listing 12-11 from within a query window, the output in Report 12-3 is returned.

Report 12-3: Output returned to application when running code in Listing 12-11

ID	ProductName	Price	ID	ProductName	Price
3	DingDong	2.99	3	DingDong	2.11

If you review the code in Listing 12-11, you will see that the OUTPUT clause contains references to both the DELETED and INSERTED pseudo-tables. By specifying the <dot>* notation following each of those pseudo tables, I told SQL Server to return the before and after images for all column values for the updated rows. By referencing both pseudo-tables, the first set of column values returned in my example represent the before image. In contrast, the second set of column values returned represent the values of my updated columns after the UPDATE statement was performed.

The values contained in these two pseudo-tables can also be placed in a table or table variable. Additionally, you don't have to output each of the pseudo-table column values either. This can be demonstrated by looking at the code in Listing 12-12.

Listing 12-12: Using the OUTPUT clause on an UPDATE statement to capture pseudo table column values in a temporary table variable

```
DECLARE @ProductPriceAudit TABLE (
   ID INT,
   BeforePrice decimal(6,2),
   AfterPrice decimal(6,2));
UPDATE Product
   SET Price = 27.98
   OUTPUT DELETED.ID,DELETED.Price,INSERTED.Price
      INTO @ProductPriceAudit
WHERE ID = 1;
SELECT * FROM @ProductPriceAudit;
```

The code in Listing 12-12 captured the before and after values of the Price column for the ID's that were updated. For a complete explanation of how to use the OUTPUT clause and its limitations, refer to the SQL Server documentation topic on the OUTPUT clause.

Maintaining Data with the UPDATE Statements

It is good to know how to use the FROM and WHERE clause to identify which rows to update and to specify the column values to use in an UPDATE statement. Being able to use the INSERTED and DELETED pseudo-tables comes in handy when you need to provide an audit trail for your update processes. Having a good understanding of how to build an UPDATE statement allows you to maintain the data in your database.

Chapter 13

How to Delete Rows from a Table

You may have data in a database that was inserted into a table by mistake, or you may have data in your tables that is no longer of value. In either case, when you have unwanted data in a table, you need a way to remove it. The DELETE statement can be used to eliminate data in a table that is no longer needed. In this chapter, I will show you different ways to use the DELETE statement to identify and remove unwanted data from your SQL Server tables.

Creating Sample Data

In order to show you how to delete records using the DELETE statement, I first need to create a couple of sample tables using the code in Listing 13-1.

Listing 13-1: Script to create sample data

```
SET NOCOUNT ON;
USE tempdb;
GO
DROP TABLE IF EXISTS Product;
CREATE TABLE dbo.Product (
      ID int identity not null,
      ProductName varchar(25) not null,
      Price decimal(6,2) not null
      );
INSERT INTO dbo.Product
      VALUES ('Widget',25.99),
             ('WingDing',18.87),
             ('DingDong',1.99),
```

```
            ('Doo-Dads',87.34),
            ('Doohickey',19.56),
            ('Thingamagigs',239.10),
            (' Whatchamacallit',3.47);
CREATE TABLE dbo.DiscontinuedProduct (
      ProductName varchar(25) not null);
INSERT INTO dbo.DiscontinuedProduct
      VALUES ('Widget');
```

The code in Listing 13-1 creates and populates two tables: *Product* and *DiscontinuedProduct*.

Deleting all the Rows in a Table

It is extremely easy to delete all the rows in a table. All you need to do is provide a DELETE statement that is not constrained. The code in Listing 13-2 deletes all the records in my *Product* table.

Listing 13-2: Deleting all rows in a table

```
USE tempdb;
GO
DELETE FROM dbo.Product;
```

You only need to be granted DELETE permissions on a table in order to use the DELETE statement to remove rows from a table.

Another way to delete all the rows in a table is to use the TRUNCATE TABLE statement. This is much more efficient in those cases where you want to remove ALL the rows from a table. Permission to run TRUNCATE TABLE cannot be granted, but the table owner and members of the sysadmin fixed server role, db_owner role, and db_ddladmin fixed database role can execute the TRUNCATE TABLE statement. A complete discussion of the TRUNCATE TABLE

statement is outside the scope of this article. If you would like more information on this statement, refer to the online documentation.

Since the code in Listing 13-2 deleted all the rows in my *Product* table, I need run the code in Listing 13-3 to repopulate this table again so it can be used for some of my other examples.

Listing 13-3: Repopulating the rows in a table

```
SET NOCOUNT ON;
USE tempdb;
GO
INSERT INTO dbo.Product
      VALUES ('Widget',25.99),
             ('WingDing',18.87),
             ('DingDong',1.99),
             ('Doo-Dads',87.34),
             ('Doohickey',19.56),
             ('Thingamagigs',239.10),
             ('Whatchamacallit',3.47);
```

Deleting Rows Based on a WHERE Condition

Most of your delete operations in your applications will probably not need to delete all the rows in a table. Instead, you will usually delete only a subset of rows based on some WHERE condition. You may want to delete a single row or multiple rows with one or more different conditions. The code in Listing 13-4 uses the WHERE clause to identify the specific rows to delete.

Listing 13-4: Deleting a Single Row

```
USE tempdb;
GO
DELETE FROM dbo.Product
WHERE ProductName = 'Thingamagigs';
```

When I run this code, only the rows that have a *ProductName* of "Thingamagigs" will be deleted. In my sample, the *Product* table has only one row meets the criteria, so only a single row was deleted.

You may find you need to delete several different rows, with each rowing have one of a list of specific values. You can do this by specifying a WHERE condition that matches multiple rows as I have done in Listing 13-5.

Listing 13-5: Use the IN clause to identify multiple rows to delete

```
USE tempdb;
GO
DELETE FROM dbo.Product
WHERE ProductName in ('DingDong','Doo-Dads','GallyWhapper');
```

In Listing 13-5, I used the IN clause to identify that I wanted to delete any *Product* row that had a *ProductName* of "DingDong", "Doo-Dads", or "GallyWhapper". When I ran this code against my sample *Product* table 2, those rows that had a *ProductName* of "DingDong" or "Doo-Dads" were deleted. Since no rows matched the "GallyWhapper", value no rows were deleted for this value. If none of the rows in my table matches any of my WHERE criteria, then no rows would be deleted. You could add as many WHERE conditions as needed to delete the rows you need.

Deleting a Specific Number of Rows

You may want only to delete a specific number of rows in a table, or only a subset of the rows that meet a particular search criterion. To handle deleting the first N rows, SQL Server allows you to use the TOP clause. The code in Listing 13-6 demonstrates using the TOP clause to delete only the first row based on my search criteria.

Listing 13-6: Deleting a single row using the TOP clause

```
USE tempdb;
GO
SELECT * FROM dbo.Product WHERE ID > 8;
DELETE TOP (1) FROM dbo.Product
WHERE ID > 8;
SELECT * FROM dbo.Product WHERE ID > 8;
```

In this example, I first run a SELECT statement that shows the rows that have an *ID* greater than 8, which, for my example, is only 2 records. Then I run a DELETE statement that will delete the first row returned where the *ID* value is greater than 8 by specifying the TOP clause. I then run a second SELECT statement to verify that only 1 row was deleted. Keep in mind that since the rows that meet my criteria are not returned in any specific sorted order, my DELETE statement only deletes the first row returned, whichever row it might be.

Identifying Rows to Delete Based on another Table

There are times when you want to delete rows in one table, based on values from another table. Listing 13-7 shows you how to use an INNER JOIN clause between two tables to identify those rows that need to be deleted.

Listing 13-7: Deleting rows based on a JOIN operation

```
USE tempdb;
GO
DELETE FROM dbo.Product
FROM dbo.Product P INNER JOIN dbo.DiscontinuedProduct D
ON P.ProductName = D.ProductName;
```

In Listing 13-7, every row in the *Product* table that meets the join criteria with the *DiscontinuedProduct* table will get deleted from the *Product* table.

Another way to delete rows in one table based on another table is to use a sub-query, as demonstrated in Listing 13-8. (If you ran the query in Listing 13-7, this one will not delete any additional rows.)

Listing 13-8: Deleting rows based on a JOIN operation

```
DELETE FROM dbo.Product
WHERE ProductName in
      (SELECT ProductName FROM dbo.DiscontinuedProduct);
```

In Listing 13-8, I identified the rows that needed to be deleted by using an IN clause within a WHERE condition. The candidate values for my IN clause were identified by using a sub-query.

Deleting Duplicate Rows using Common Table Expression

The definition of duplicate rows may vary, but for this discussion, a duplicate row is a row that has the same column value for each and every column as another row in a table. How do you go about deleting duplicate rows? There are several options for doing that. Each method needs a way to uniquely identify each duplicate row so you can delete all but one of the duplicate rows. For my example on how to delete duplicate rows, I will show you how to uniquely identify and delete duplicate rows using a common table expression (CTE).

The code in Listing 13-9 creates some sample data that I will be using to demonstrate how to delete duplicate rows.

Listing 13-9: Creating sample data

```
SET NOCOUNT ON;
USE tempdb;
GO
CREATE TABLE dbo.Dups (Id int, Name varchar(10));
INSERT INTO Dups VALUES (1,'Red');
INSERT INTO Dups VALUES (2,'White');
INSERT INTO Dups VALUES (2,'White');
INSERT INTO Dups VALUES (3,'Blue');
INSERT INTO Dups VALUES (3,'Blue');
INSERT INTO Dups VALUES (3,'Blue');
SELECT * FROM Dups;
```

CTEs are an advanced topic and a complete discussion of what they are and how to use them are outside the scope of this chapter. Think of a CTE as a pseudo table where the rows are created from the T-SQL code contained in the CTE. In Listing 13-10, you will find my example that uses a CTE to delete the duplicate rows from my sample data table *Dups*.

Listing 13-10: Displaying duplicate rows using a CTE

```
USE tempdb;
GO
WITH DupRecords (Id,Name, DuplicateCount)
AS
(
SELECT Id, Name,
ROW_NUMBER() OVER(PARTITION BY Id, Name
                  ORDER BY Id,Name) AS DuplicateCount
FROM dbo.Dups
)
SELECT * FROM DupRecords;
```

The code in Listing 13-10 first defines the CTE named *DupRecords*. This CTE returns a result set that contains all the rows in the *dbo.Dups* table, and then also includes a column named *DuplicateCount*. The *DuplicateCount* column is created using the ROW_NUMBER() function, which sequentially numbers each row based on the *Name* and *Id* column values. The results of ROW_NUMBER() function numbers each duplicate *Name* and *Id* value row with a different number starting with 1 for the first row, 2 for the second row with the same *Name* and *Id* value, and so on. The code in Listing 13-10 then takes the defined CTE and uses it in a SELECT statement. By running this code, you will be able to see how this CTE creates a recordset that contains a *DuplicateCount* column value. By looking at the output of this CTE, you can see any time the row is a duplicate value, the duplicate rows are numbered sequentially, and the sequential number is the *DuplicateCount* column.

The code in Listing 13-11 defines the same CTE as in Listing 13-10, but this time uses it in a DELETE statement.

Listing 13-11: Deleting duplicate records using a CTE

```
USE tempdb;
GO
WITH DupRecords (Id,Name, DuplicateCount)
AS
(
SELECT Id, Name,
ROW_NUMBER() OVER(PARTITION BY Id, Name
                  ORDER BY Id,Name) AS DuplicateCount
FROM dbo.Dups
)
 DELETE
FROM DupRecords
WHERE DuplicateCount > 1
GO
SELECT * FROM dbo.Dups;
```

The DELETE statement uses the CTE to remove any rows that have a *DuplicateCount* > 1, which essentially deletes the rows with duplicate color values from the *dbo.Dups* table. Note a CTEs can only be used in the statement immediately following the CTE declaration. This is why I defined the CTE a second time.

Using the Output Clause

Have you ever wondered what you just deleted with a DELETE statement? If you have, then there is help for you by using the OUTPUT clause. When you use the OUTPUT clause on your DELETE statement, information about the deleted row is stored in a pseudo-table named DELETED.

Let me go through a couple of examples to show you how this works. But first, let me create and populate a new table in Listing 13-12 that will be used in my follow-on DELETE statements.

Listing 13-12: Create a sample table

```
SET NOCOUNT ON;
USE tempdb;
GO
CREATE TABLE dbo.MyVehicles (ID int,
                 Name varchar(10),
                 Color varchar(10));
INSERT INTO dbo.MyVehicles values (1,'Volvo','Green');
INSERT INTO dbo.MyVehicles values (2,'Kia','Black');
INSERT INTO dbo.MyVehicles values (3,'Chevrolet','Gold');
INSERT INTO dbo.MyVehicles values (4,'Nissan','Red');
INSERT INTO dbo.MyVehicles values (5,'Honda','Gold');
```

Suppose I want to remove all my vehicles that are of ”Gold” color, and return the column values for every row deleted to my application. All I would need to do is run the code in Listing 13-13:

Listing 13-13: Outputting deleted column/row values for deleted rows

```
USE tempdb;
GO
DELETE FROM dbo.MyVehicles
      OUTPUT DELETED.*
WHERE Color = 'Gold';
```

The code in Listing 13-13 returns all the column values for each row deleted. It does this because I specified “DELETED.*” in the OUTPUT clause which means to return all deleted

column values. When I run the code in Listing 13-13 from a Query Window, I get the output found in Report 13-1.

Report 13-1: Output from running the code in Listing 13-13

ID	Name	Color
3	Chevrolet	Gold
5	Honda	Gold

By reviewing the output, you can tell 2 vehicles were deleted that had a color of “Gold”.

You don’t have to specify that every column of the deleted rows needs to be outputted. Additionally, you don’t have to return the deleted row information to your application. Instead, you can write the output for the deleted rows to a table. To show you how this works, I have selectively outputted columns to a table in my DELETE statement using the code in Listing 13-14.

Listing 13-14: Outputting only the *Name* and *Color* column to a table variable

```
USE tempdb;
GO
DECLARE @DeletedValues TABLE (Name varchar(10),
                        Color varchar(10));

DELETE FROM dbo.MyVehicles
      OUTPUT DELETED.Name, DELETED.Color
      INTO @DeletedValues
WHERE Color in ('Black','Red');
SELECT * FROM @DeletedValues;
```

The code in Listing 13-14 identified only two columns to be outputted, *Name* and *Color*. I also include the INTO clause which caused the deleted column values for each deleted row to be

inserted into my table variable *@DeletedValues*. By doing this, I was able to run a SELECT statement right after the DELETE statement that displayed the column values for the deleted rows that were inserted into my table variable.

Building Blocks for Deleting Rows

Deleting data from a table is needed when you mistakenly enter data into a table, or the data is no longer of any value. Being able to delete all the data in a table or individual rows is something that most applications will occasionally require. Care should be taken when deleting data to make sure you don't delete too much or not enough data. Hopefully, this article provided you with the building blocks necessary to allow you to build your own DELETE statements to remove the rows you no longer need in your database tables.

Transact-SQL

The Building Blocks to SQL Server Programming

By Gregory A. Larsen

Transact SQL (TSQL) code is used to maintain and unlock the knowledge of data stored in a SQL Server. The TSQL language is a proprietary version of the SQL language developed by Microsoft. TSQL is used to maintain, manipulate and report on data stored in SQL Server. This book will cover different aspects of the TSQL language. With an understanding the basics of the TSQL language, programmers will have the building blocks necessary to quickly and easily build applications that use SQL Server.

This book takes the readers step by step through the TSQL language, one concept at a time. Each topic covers a particular aspect of TSQL with discussion and examples. The basics of the TSQL language are explored first, by starting with the SELECT statement. As the reader progresses through each chapter, they will learn about how to, join, sort, group, and aggregate data. In addition to the basic programming topics, this book also covers the history of SQL Server and basics of set theory. By the end of the book, the readers will have a good foundation of how to write SELECT, UPDATE, INSERT and DELETE statements to manipulate SQL Server data.

For further expert SQL Server content visit www.redgate.com/Hub

redgate