Pro T-SQL 2008 Programmer's Guide

ISBN-13 (pbk): 978-1-4302-1001-6

ISBN-13 (electronic): 978-1-4302-1002-3

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editors: Jonathan Gennick, Tony Campbell
Technical Reviewer: Adam Machanic
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editor: Damon Larson
Associate Production Director: Kari Brooks-Copony
Production Editor: Elizabeth Berry
Compositor: Lynn L'Heureux
Proofreaders: Linda Seifert, April Eddy
Indexer: Broccoli Information Management
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 1

Foundations of T-SQL

SQL Server 2008 is the latest release of Microsoft's enterprise-class database management system (DBMS). As the name implies, a DBMS is a tool designed to manage, secure, and provide access to data stored in structured collections within databases. T-SQL is the language that SQL Server speaks. T-SQL provides query and data manipulation functionality, data definition and management capabilities, and security administration tools to SQL Server developers and administrators. To communicate effectively with SQL Server, you must have a solid understanding of the language. In this chapter, we will begin exploring T-SQL on SQL Server 2008.

A Short History of T-SQL

The history of Structured Query Language (SQL), and its direct descendant Transact-SQL (T-SQL), begins with a man. Specifically, it all began in 1970 when Dr. E. F. Codd published his influential paper "A Relational Model of Data for Large Shared Data Banks" in the Communications of the Association for Computing Machinery (ACM). In his seminal paper, Dr. Codd introduced the definitive standard for relational databases. IBM went on to create the first relational database management system, known as System R. They subsequently introduced the Structured English Query Language (SEQUEL, as it was known at the time) to interact with this early database to store, modify, and retrieve data. The name of this early query language was later changed from SEQUEL to the now-common SQL due to a trademark issue.

Fast-forward to 1986 when the American National Standards Institute (ANSI) officially approved the first SQL standard, commonly known as the ANSI SQL-86 standard. Microsoft entered the relational database management system picture a few years later through a joint venture with Sybase and Ashton-Tate (of dBase fame). The original versions of Microsoft SQL Server shared a common code base with the Sybase SQL Server product. This changed with the release of SQL Server 7.0, when Microsoft partially rewrote the code base. Microsoft has since introduced several iterations of SQL Server, including SQL Server 2000, SQL Server 2005, and now SQL Server 2008. In this book, we will focus on SQL Server 2008, which further extends the capabilities of T-SQL beyond what was possible in previous releases.

Imperative vs. Declarative Languages

SQL is different from many common programming languages such as C# and Visual Basic because it is a *declarative language*. To contrast, languages such as C++, Visual Basic, C#, and even assembler language are *imperative languages*. The imperative language model requires

the user to determine what the end result should be and also tell the computer step by step how to achieve that result. It's analogous to asking a cab driver to drive you to the airport, and then giving him turn-by-turn directions to get there. Declarative languages, on the other hand, allow you to frame your instructions to the computer in terms of the end result. In this model, you allow the computer to determine the best route to achieve your objective, analogous to just telling the cab driver to take you to the airport and trusting him to know the best route. The declarative model makes a lot of sense when you consider that SQL Server is privy to a lot of "inside information." Just like the cab driver who knows the shortcuts, traffic conditions, and other factors that affect your trip, SQL Server inherently knows several methods to optimize your queries and data manipulation operations.

Consider Listing 1-1, which is a simple C# code snippet that reads in a flat file of names and displays them on the screen.

Listing 1-1. *C# Snippet to Read a Flat File*

```
StreamReader sr = new StreamReader("c:\\Person_Person.txt");
string FirstName = null;
while ((FirstName = sr.ReadLine()) != null) {
    Console.WriteLine(s);
}
sr.Dispose();
```

The example performs the following functions in an orderly fashion:

1. The code explicitly opens the storage for input (in this example, a flat file is used as a "database").
2. It then reads in each record (one record per line), explicitly checking for the end of the file.
3. As it reads the data, the code returns each record for display using `Console.WriteLine()`.
4. And finally, it closes and disposes of the connection to the data file.

Consider what happens when you want to add or delete a name from the flat-file "database." In those cases, you must extend the previous example and add custom routines to explicitly reorganize all the data in the file so that it maintains proper ordering. If you want the names to be listed and retrieved in alphabetical (or any other) order, you must write your own sort routines as well. Any type of additional processing on the data requires that you implement separate procedural routines.

The SQL equivalent of the C# code in Listing 1-1 might look something like Listing 1-2.

Listing 1-2. *SQL Query to Retrieve Names from a Table*

```
SELECT FirstName
FROM Person.Person;
```

■ **Tip** Unless otherwise specified, you can run all the T-SQL samples in this book in the AdventureWorks 2008 sample database using SQL Server Management Studio or SQLCMD.

To sort your data, you can simply add an `ORDER BY` clause to the `SELECT` query in Listing 1-2. With properly designed and indexed tables, SQL Server can automatically reorganize and index your data for efficient retrieval after you insert, update, or delete rows.

T-SQL includes extensions that allow you to use procedural syntax. In fact, you could rewrite the previous example as a cursor to closely mimic the C# sample code. These extensions should be used with care, however, since trying to force the imperative model on T-SQL effectively overrides SQL Server's built-in optimizations. More often than not, this hurts performance and makes simple projects a lot more complex than they need to be.

One of the great assets of SQL Server is that you can invoke its power, in its native language, from nearly any other programming language. For example, in .NET you can connect and issue SQL queries and T-SQL statements to SQL Server via the `System.Data.SqlClient` namespace, which I will discuss further in Chapter 15. This gives you the opportunity to combine SQL's declarative syntax with the strict control of an imperative language.

SQL Basics

Before we discuss development in T-SQL, or on any SQL-based platform for that matter, we have to make sure we're speaking the same language. Fortunately for us, SQL can be described accurately using well-defined and time-tested concepts and terminology. We'll begin our discussion of the components of SQL by looking at *statements*.

Statements

To begin with, in SQL we use statements to communicate our requirements to the DBMS. A statement is composed of several parts, as shown in Figure 1-1.

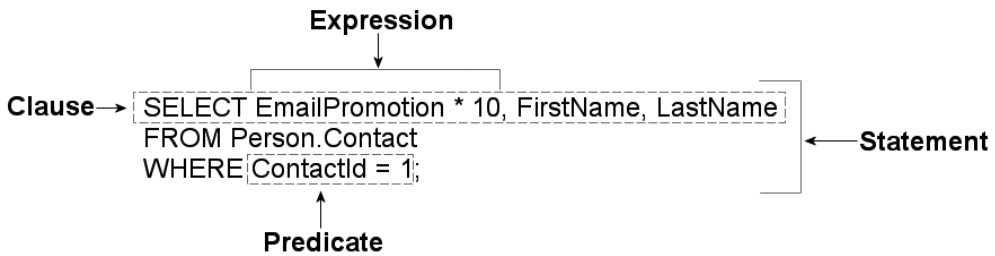

Figure 1-1. *Components of a SQL statement*

As you can see in the figure, SQL statements are composed of one or more *clauses*, some of which may be optional depending on the statement. In the `SELECT` statement shown, there are three clauses: the `SELECT` clause, which defines the columns to be returned by the query; the `FROM` clause, which indicates the source table for the query; and the `WHERE` clause, which is used to limit the results. Each clause represents a primitive operation in the relational algebra. For instance, in the example, the `SELECT` clause represents a relational *projection* operation, the `FROM` clause indicates the *relation*, and the `WHERE` clause performs a *restriction* operation.

■ **Note** The *relational model* of databases is the model formulated by Dr. E. F. Codd. In the relational model, what we know in SQL as *tables* are referred to as *relations*; hence the name. *Relational calculus* and *relational algebra* define the basis of query languages for the relational model in mathematical terms.

ORDER OF EXECUTION

Understanding the logical order in which SQL clauses are applied within a statement or query is important when setting your expectations about results. While vendors are free to physically perform whatever operations, in any order, that they choose to fulfill a query request, the results must be the same as if the operations were applied in a standards-defined order.

The `WHERE` clause in the example contains a *predicate*, which is a logical expression that evaluates to one of SQL's three possible logical results: true, false, or unknown. In this case, the `WHERE` clause and the predicate limit the results returned so that they include only rows in which the `ContactId` column is equal to 1.

The `SELECT` clause includes an expression that is calculated during statement execution. In the example, the expression `EmailPromotion * 10` is used. This expression is calculated for every row of the result set.

SQL THREE-VALUED LOGIC

SQL institutes a logic system that might seem foreign to developers coming from other languages like C++ or Visual Basic (or most other programming languages, for that matter). Most modern computer languages use simple two-valued logic: a Boolean result is either true or false. SQL supports the concept of `NULL`, which is a placeholder for a missing or unknown value. This results in a more complex three-valued logic (3VL).

Let me give you a quick example to demonstrate. If I asked you the question, "Is *x* less than 10?" your first response might be along the lines of, "How much is *x*?" If I refused to tell you what value *x* stood for, you would have no idea whether *x* was less than, equal to, or greater than 10; so the answer to the question is neither true nor false—it's the third truth value, *unknown*. Now replace *x* with `NULL` and you have the essence of SQL 3VL. `NULL` in SQL is just like a variable in an equation when you don't know the variable's value.

No matter what type of comparison you perform with a missing value, or which other values you compare the missing value to, the result is always unknown. I'll continue the discussion of SQL 3VL in Chapter 4.

The core of SQL is defined by statements that perform five major functions: querying data stored in tables, manipulating data stored in tables, managing the structure of tables, controlling access to tables, and managing transactions. All of these subsets of SQL are defined following:

- *Querying*: The `SELECT` query statement is a complex statement. It has more optional clauses and vendor-specific tweaks than any other statement, bar none. `SELECT` is concerned simply with retrieving data stored in the database.
- *Data Manipulation Language (DML)*: DML is considered a sublanguage of SQL. It is concerned with manipulating data stored in the database. DML consists of four commonly used statements: `INSERT`, `UPDATE`, `DELETE`, and `MERGE`. DML also encompasses cursor-related statements. These statements allow you to manipulate the contents of tables and persist the changes to the database.
- *Data Definition Language (DDL)*: DDL is another sublanguage of SQL. The primary purpose of DDL is to create, modify, and remove tables and other objects from the database. DDL consists of variations of the `CREATE`, `ALTER`, and `DROP` statements.
- *Data Control Language (DCL)*: DCL is yet another SQL sublanguage. DCL's goal is to allow you to restrict access to tables and database objects. DCL is composed of various `GRANT` and `REVOKE` statements that allow or deny users access to database objects.
- *Transactional Control Language (TCL)*: TCL is the SQL sublanguage that is concerned with initiating and committing or rolling back *transactions*. A transaction is basically an atomic unit of work performed by the server. The `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK` statements comprise TCL.

Databases

A SQL Server *instance*—an individual installation of SQL Server with its own ports, logins, and databases—can manage multiple *system databases* and *user databases*. SQL Server has five system databases, as follows:

- `resource`: The `resource` database is a read-only system database that contains all system objects. You will not see the `resource` database in the SQL Server Management Studio (SSMS) Object Explorer window, but the system objects persisted in the `resource` database will logically appear in every database on the server.
- `master`: The `master` database is a server-wide repository for configuration and status information. The `master` database maintains instance-wide metadata about SQL Server as well as information about all databases installed on the current instance. It is wise to avoid modifying or even accessing the `master` database directly in most cases. An entire server can be brought to its knees if the `master` database is corrupted. If you need to access the server configuration and status information, use catalog views instead.
- `model`: The `model` database is used as the template from which newly created databases are essentially cloned. Normally, you won't want to change this database in production settings, unless you have a very specific purpose in mind and are extremely knowledgeable about the potential implications of changing the `model` database.

- `msdb`: The `msdb` database stores system settings and configuration information for various support services, such as SQL Agent and Database Mail. Normally, you will use the supplied stored procedures and views to modify and access this data, rather than modifying it directly.
- `tempdb`: The `tempdb` database is the main working area for SQL Server. When SQL Server needs to store intermediate results of queries, for instance, they are written to `tempdb`. Also, when you create temporary tables, they are actually created within `tempdb`. The `tempdb` database is reconstructed from scratch every time you restart SQL Server.

Microsoft recommends that you use the system-provided stored procedures and catalog views to modify system objects and system metadata, and let SQL Server manage the system databases itself. You should avoid modifying the contents and structure of the system databases directly.

User databases are created by database administrators (DBAs) and developers on the server. These types of databases are so called because they contain user data. The AdventureWorks 2008 sample database is one example of a user database.

Transaction Logs

Every SQL Server database has its own associated transaction log. The transaction log provides recoverability in the event of failure, and ensures the atomicity of transactions. The transaction log accumulates all changes to the database so that database integrity can be maintained in the event of an error or other problem. Because of this arrangement, all SQL Server databases consist of at least two files: a database file with an `.mdf` extension and a transaction log with an `.ldf` extension.

THE ACID TEST

SQL folks, and IT professionals in general, love their acronyms. A common acronym in the SQL world is ACID, which stands for "atomicity, consistency, isolation, durability." These four words form a set of properties that database systems should implement to guarantee reliability of data storage, processing, and manipulation.

- *Atomicity*: All data changes should be transactional in nature. That is, data changes should follow an all-or-nothing pattern. The classic example is a double-entry bookkeeping system in which every debit has an associated credit. Recording a debit-and-credit double-entry in the database is considered one "transaction," or a single unit of work. You cannot record a debit without recording its associated credit, and vice versa. Atomicity ensures that either the entire transaction is performed or none of it is.
- *Consistency*: Only data that is consistent with the rules set up in the database will be stored. Data types and constraints can help enforce consistency within the database. For instance, you cannot insert the name Dolly in an `int` column. Consistency also applies when dealing with data updates. If two users update the same row of a table at the same time, an inconsistency could occur if one update is only partially complete when the second update begins. The concept of isolation, described following, is designed to deal with this situation.

- *Isolation*: Multiple simultaneous updates to the same data should not interfere with one another. SQL Server includes several locking mechanisms and isolation levels to ensure that two users cannot modify the exact same data at the exact same time, which could put the data in an inconsistent state. Isolation also prevents you from even reading uncommitted data by default.
- *Durability*: Data that passes all the previous tests is committed to the database. The concept of durability ensures that committed data is not lost. The transaction log and data backup and recovery features help to ensure durability.

The transaction log is one of the main tools SQL Server uses to enforce the ACID concept when storing and manipulating data.

Schemas

SQL Server 2008 supports database schemas, which are little more than logical groupings of database objects. The AdventureWorks 2008 sample database, for instance, contains several schemas, such as `HumanResources`, `Person`, and `Production`. These schemas are used to group tables, stored procedures, views, and user-defined functions (UDFs) for management and security purposes.

■ **Tip** When you create new database objects, like tables, and don't specify a schema, they are automatically created in the default schema. The default schema is normally `dbo`, but DBAs may assign different default schemas to different users. Because of this, it's always best to specify the schema name explicitly when creating database objects.

Tables

SQL Server supports several types of objects that can be created within a database. SQL stores and manages data in its primary data structures, tables. A table consists of rows and columns, with data stored at the intersections of these rows and columns. As an example, the AdventureWorks `HumanResources.Department` table is shown in Figure 1-2.

In the table, each row is associated with columns and each column has certain restrictions placed on its content. These restrictions comprise the *data domain*. The data domain defines all the values a column can contain. At the lowest level, the data domain is based on the data type of the column. For instance, a `smallint` column can contain any integer values between –32,768 and +32,767.

The data domain of a column can be further constrained through the use of check constraints, triggers, and foreign key constraints. *Check constraints* provide a means of automatically checking that the value of a column is within a certain range or equal to a certain value whenever a row is inserted or updated. *Triggers* can provide similar functionality to check constraints. *Foreign key constraints* allow you to declare a relationship between the columns of one table and the columns of another table. You can use foreign key constraints to restrict the data domain of a column to only include those values that appear in a designated column of another table.

Departmen...	Name	GroupName	ModifiedDate
1	Engineering	Research and Development	1998-06-01 00:00:00.000
2	Tool Design	Research and Development	1998-06-01 00:00:00.000
3	Sales	Sales and Marketing	1998-06-01 00:00:00.000
4	Marketing	Sales and Marketing	1998-06-01 00:00:00.000
5	Purchasing	Inventory Management	1998-06-01 00:00:00.000
6	Research and Development	Research and Development	1998-06-01 00:00:00.000
7	Production	Manufacturing	1998-06-01 00:00:00.000
8	Production Control	Manufacturing	1998-06-01 00:00:00.000
9	Human Resources	Executive General and Ad...	1998-06-01 00:00:00.000
10	Finance	Executive General and Ad...	1998-06-01 00:00:00.000
11	Information Services	Executive General and Ad...	1998-06-01 00:00:00.000
12	Document Control	Quality Assurance	1998-06-01 00:00:00.000
13	Quality Assurance	Quality Assurance	1998-06-01 00:00:00.000
14	Facilities and Maintenance	Executive General and Ad...	1998-06-01 00:00:00.000
15	Shipping and Receiving	Inventory Management	1998-06-01 00:00:00.000
16	Executive	Executive General and Ad...	1998-06-01 00:00:00.000

Figure 1-2. *Representation of the HumanResources.Department table*

RESTRICTING THE DATA DOMAIN: A COMPARISON

In this section, I have given a brief overview of three methods of constraining the data domain for a column—restricting the values that can be contained in the column. Here's a quick comparison of the three methods:

- Foreign key constraints allow SQL Server to perform an automatic check against another table to ensure that the values in a given column exist in the referenced table. If the value you are trying to update or insert in a table does not exist in the referenced table, an error is raised. The foreign key constraint provides a flexible means of altering the data domain, since adding or removing values from the referenced table automatically changes the data domain for the referencing table. Also, foreign key constraints offer an additional feature known as cascading *declarative referential integrity (DRI)*, which automatically updates or deletes rows from a referencing table if an associated row is removed from the referenced table.
- Check constraints provide a simple, efficient, and effective tool for ensuring that the values being inserted or updated in a column are within a given range or a member of a given set of values. Check constraints, however, are not as flexible as foreign key constraints and triggers since the data domain is normally defined using hard-coded constant values.
- Triggers are stored procedures attached to insert, update, or delete events on a table. A trigger provides a flexible solution for constraining data, but it may require more maintenance than the other options since it is essentially a specialized form of stored procedure. Unless they are extremely well designed, triggers have the potential to be much less efficient than the other methods, as well. Triggers to constrain the data domain are generally avoided in modern databases in favor of the other methods. The exception to this is when you are trying to enforce a foreign key constraint across databases, since SQL Server doesn't support cross-database foreign key constraints.

Which method you use to constrain the data domain of your column(s) needs to be determined by your project-specific requirements on a case-by-case basis.

Views

A view is like a virtual table—the data it exposes is not stored in the view object itself. Views are composed of SQL queries that reference tables and other views, but they are referenced just like tables in queries. Views serve two major purposes in SQL Server: they can be used to hide the complexity of queries, and they can be used as a security device to limit the rows and columns of a table that a user can query. Views are expanded, meaning that their logic is incorporated into the execution plan for queries when you use them in queries and DML statements. SQL Server may not be able to use indexes on the base tables when the view is expanded, resulting in less-than-optimal performance when querying views in some situations.

To overcome the query performance issues with views, SQL Server also has the ability to create a special type of view known as an *indexed view*. An indexed view is a view that SQL Server persists to the database like a table. When you create an indexed view, SQL Server allocates storage for it and allows you to query it like any other table. There are, however, restrictions on inserting, updating, and deleting from an indexed view. For instance, you cannot perform data modifications on an indexed view if more than one of the view's base tables will be affected. You also cannot perform data modifications on an indexed view if the view contains aggregate functions or a `DISTINCT` clause.

You can also create indexes on an indexed view to improve query performance. The downside to an indexed view is increased overhead when you modify data in the view's base tables, since the view must be updated as well.

Indexes

Indexes are SQL Server's mechanisms for optimizing access to data. SQL Server 2008 supports several types of indexes, including the following:

- *Clustered index*: A clustered index is limited to one per table. This type of index defines the ordering of the rows in the table. A clustered index is physically implemented using a b-tree structure with the data stored in the leaf levels of the tree. Clustered indexes order the data in a table in much the same way that a phone book is ordered by last name. A table with a clustered index is referred to as a *clustered table*, while a table with no clustered index is referred to as a *heap*.
- *Nonclustered index*: A nonclustered index is also a b-tree index managed by SQL Server. In a nonclustered index, *index rows* are included in the leaf levels of the b-tree. Because of this, nonclustered indexes have no effect on the ordering of rows in a table. The index rows in the leaf levels of a nonclustered index consist of the following:
  - A nonclustered key value
  - A row locator, which is the clustered index key on a table with a clustered index, or a SQL-generated row ID for a heap
  - Nonkey columns, which are added via the `INCLUDE` clause of the `CREATE INDEX` statement

  A nonclustered index is analogous to an index in the back of a book.

- *XML index*: SQL Server supports special indexes designed to help efficiently query XML data. See Chapter 11 for more information.
- *Spatial index*: A spatial index is an interesting new indexing structure to support efficient querying of the new geometry and geography data types. See Chapter 2 for more information.
- *Full-text index*: A full-text index (FTI) is a special index designed to efficiently perform full-text searches of data and documents.

Beginning with SQL Server 2005, you can also include nonkey columns in your nonclustered indexes with the `INCLUDE` clause of the `CREATE INDEX` statement. The included columns give you the ability to work around SQL Server's index size limitations.

Stored Procedures

SQL Server supports the installation of server-side T-SQL code modules via *stored procedures (SPs)*. It's very common to use SPs as a sort of intermediate layer or custom server-side *application programming interface (API)* that sits between user applications and tables in the database. Stored procedures that are specifically designed to perform queries and DML statements against the tables in a database are commonly referred to as *CRUD (create, read, update, delete)* procedures.

User-Defined Functions

User-defined functions (UDFs) can perform queries and calculations, and return either scalar values or tabular result sets. UDFs have certain restrictions placed on them. For instance, they cannot utilize certain nondeterministic system functions, nor can they perform DML or DDL statements, so they cannot make modifications to the database structure or content. They cannot perform dynamic SQL queries or change the state of the database (i.e., cause side effects).

SQL CLR Assemblies

SQL Server 2008 supports access to Microsoft .NET functionality via the SQL Common Language Runtime (SQL CLR). To access this functionality, you must register compiled .NET SQL CLR assemblies with the server. The assembly exposes its functionality through class methods, which can be accessed via SQL CLR functions, procedures, triggers, user-defined types, and user-defined aggregates. SQL CLR assemblies replace the deprecated SQL Server extended stored procedure (XP) functionality available in prior releases.

■ **Tip** Avoid using XPs on SQL Server 2008. The same functionality provided by XPs can be provided by SQL CLR code. The SQL CLR model is more robust and secure than the XP model. Also keep in mind that the XP library is deprecated and XP functionality may be completely removed in a future version of SQL Server.

Elements of Style

Now that I've given a broad overview of the basics of SQL Server, we'll take a look at some recommended development tips to help with code maintenance. Selecting a particular style and using it consistently helps immensely with both debugging and future maintenance. The following sections contain some general recommendations to make your T-SQL code easy to read, debug, and maintain.

Whitespace

SQL Server ignores extra whitespace between keywords and identifiers in SQL queries and statements. A single statement or query may include extra spaces and tab characters, and can even extend across several lines. You can use this knowledge to great advantage. Consider Listing 1-3, which is adapted from the `HumanResources.vEmployee` view in the AdventureWorks database.

Listing 1-3. *The HumanResources.vEmployee View from the AdventureWorks Database*

```
SELECT [HumanResources].[Employee].[EmployeeID],[Person].[Contact].[Title],[➥
Person].[Contact].[FirstName],[Person].[Contact].[MiddleName],[Person].[Cont➥
act].[LastName],[Person].[Contact].[Suffix],[HumanResources].[Employee].[Tit➥
le] AS [JobTitle],[Person].[Contact].[Phone],[Person].[Contact].[EmailAddres➥
s],[Person].[Contact].[EmailPromotion],[Person].[Address].[AddressLine1],[Pe➥
rson].[Address].[AddressLine2],[Person].[Address].[City],[Person].[StateProv➥
ince].[Name] AS [StateProvinceName],[Person].[Address].[PostalCode],[Person]➥
.[CountryRegion].[Name] AS [CountryRegionName],[Person].[Contact].[Additiona➥
lContactInfo] FROM [HumanResources].[Employee] INNER JOIN [Person].[Contact]➥
 ON [Person].[Contact].[ContactID] = [HumanResources].[Employee].[ContactID]➥
 INNER JOIN [HumanResources].[EmployeeAddress] ON [HumanResources].[Employee➥
].[EmployeeID] = [HumanResources].[EmployeeAddress].[EmployeeID] INNER JOIN ➥
[Person].[Address] ON [HumanResources].[EmployeeAddress].[AddressID] = [Pers➥
on].[Address].[AddressID] INNER JOIN [Person].[StateProvince] ON [Person].[S➥
tateProvince].[StateProvinceID] = [Person].[Address].[StateProvinceID] INNER➥
 JOIN [Person].[CountryRegion] ON [Person].[CountryRegion].[CountryRegionCod➥
e] = [Person].[StateProvince].[CountryRegionCode]
```

This query will run and return the correct result, but it's very hard to read. You can use whitespace and table aliases to generate a version that is much easier on the eyes, as demonstrated in Listing 1-4.

Listing 1-4. *The HumanResources.vEmployee View Reformatted for Readability*

```
SELECT
    e.EmployeeID,
    c.Title,
    c.FirstName,
    c.MiddleName,
    c.LastName,
    c.Suffix,
    e.Title AS JobTitle,
    c.Phone,
    c.EmailAddress,
    c.EmailPromotion,
    a.AddressLine1,
    a.AddressLine2,
    a.City,
    sp.Name AS StateProvinceName,
    a.PostalCode,
    cr.Name AS CountryRegionName,
    c.AdditionalContactInfo
FROM HumanResources.Employee e
INNER JOIN Person.Contact c
    ON c.ContactID = e.ContactID
INNER JOIN HumanResources.EmployeeAddress ea
    ON e.EmployeeID = ea.EmployeeID
INNER JOIN Person.Address a
    ON ea.AddressID = a.AddressID
INNER JOIN Person.StateProvince sp
    ON sp.StateProvinceID = a.StateProvinceID
INNER JOIN Person.CountryRegion cr
    ON cr.CountryRegionCode = sp.CountryRegionCode;
```

Notice that the `ON` keywords are indented, associating them visually with the `INNER JOIN` operators directly before them in the listing. The column names on the lines directly after the `SELECT` keyword are also indented, associating them visually with the `SELECT` keyword. This particular style is useful in helping visually break up a query into sections. The personal style you decide upon might differ from this one, but once you have decided on a standard indentation style, be sure to apply it consistently throughout your code.

Code that is easy to read is easier to debug and maintain. The code in Listing 1-4 uses table aliases, plenty of whitespace, and the semicolon (`;`) terminator marking the end of the `SELECT` statement to make the code more readable. Although not always required, it is a good idea to get into the habit of using the terminating semicolon in your SQL queries.

■ **Note** Semicolons are required terminators for some statements in SQL Server 2008. Instead of trying to remember all the special cases where they are or aren't required, it is a good idea to use the semicolon statement terminator throughout your T-SQL code. You will notice the use of semicolon terminators in all the examples in this book.

Naming Conventions

SQL Server allows you to name your database objects (tables, views, procedures, and so on) using just about any combination of up to 128 characters (116 characters for local temporary table names), as long as you enclose them in double quotes (") or brackets ([ ]). Just because you *can*, however, doesn't necessarily mean you *should*. Many of the allowed characters are hard to differentiate from other similar-looking characters, and some might not port well to other platforms. The following suggestions will help you avoid potential problems:

- Use alphabetic characters (A–Z, a–z, and Unicode Standard 3.2 letters) for the first character of your identifiers. The obvious exceptions are SQL Server variable names that start with the at sign (@), temporary tables and procedures that start with the number sign (#), and global temporary tables and procedures that begin with a double number sign (##).
- Many built-in T-SQL functions and system variables have names that begin with a double at sign (@@), such as `@@ERROR` and `@@IDENTITY`. To avoid confusion and possible conflicts, don't use a leading double at sign to name your identifiers.
- Restrict the remaining characters in your identifiers to alphabetic characters (A–Z, a–z, and Unicode Standard 3.2 letters), numeric digits (0–9), and the underscore character (_). The dollar sign ($) character, while allowed, is not advisable.
- Avoid embedded spaces, punctuation marks (other than the underscore character), and other special characters in your identifiers.
- Avoid using SQL Server 2008 reserved keywords as identifiers.
- Limit the length of your identifiers. Thirty-two characters or less is a reasonable limit while not being overly restrictive. Much more than that becomes cumbersome to type and can hurt your code readability.

Finally, to make your code more readable, select a capitalization style for your identifiers and code, and use it consistently. My preference is to fully capitalize T-SQL keywords and use mixed-case and underscore characters to visually "break up" identifiers into easily readable words. Using all capital characters or inconsistently applying mixed case to code and identifiers can make your code illegible and hard to maintain. Consider the example query in Listing 1-5.

Listing 1-5. *All-Capital SELECT Query*

```
SELECT I.CUSTOMERID, C.TITLE, C.FIRSTNAME, C.MIDDLENAME,
    C.LASTNAME, C.SUFFIX, C.PHONE, C.EMAILADDRESS,
    C.EMAILPROMOTION
FROM SALES.INDIVIDUAL I
INNER JOIN PERSON.CONTACT C
    ON C.CONTACTID = I.CONTACTID
INNER JOIN SALES.CUSTOMERADDRESS CA
    ON CA.CUSTOMERID = I.CUSTOMERID;
```

The all-capital version is difficult to read. It's hard to tell the SQL keywords from the column and table names at a glance. Compound words for column and table names are not easily identified. Basically, your eyes have to work a lot harder to read this query than they should, which makes otherwise simple maintenance tasks more difficult. Reformatting the code and identifiers makes this query much easier on the eyes, as Listing 1-6 demonstrates.

Listing 1-6. *Reformatted, Easy-on-the-Eyes Query*

```
SELECT
    i.CustomerID,
    c.Title,
    c.FirstName,
    c.MiddleName,
    c.LastName,
    c.Suffix,
    c.Phone,
    c.EmailAddress,
    c.EmailPromotion
FROM Sales.Individual i
INNER JOIN Person.Contact c
    ON c.ContactID = i.ContactID
INNER JOIN Sales.CustomerAddress ca
    ON ca.CustomerID = i.CustomerID;
```

The use of all capitals for the keywords in the second version makes them stand out from the mixed-case table and column names. Likewise, the mixed-case column and table names make the compound word names easy to recognize. The net effect is that the code is easier to read, which makes it easier to debug and maintain. Consistent use of good formatting habits helps keep trivial changes trivial and makes complex changes easier.

One Entry, One Exit

When writing SPs and UDFs, it's good programming practice to use the "one entry, one exit" rule. SPs and UDFs should have a single entry point and a single exit point (`RETURN` statement). The following SP retrieves the `ContactTypeID` number from the AdventureWorks `Person.ContactType` table for the `ContactType` name passed into it. If no `ContactType` exists with the name passed in, a new one is created, and the newly created `ContactTypeID` is passed back. Listing 1-7 demonstrates this simple procedure with one entry point and several exit points.

Listing 1-7. *Stored Procedure Example with One Entry and Multiple Exits*

```
CREATE PROCEDURE dbo.GetOrAdd_ContactType
(
    @Name NVARCHAR(50),
    @ContactTypeID INT OUTPUT
)
AS
    DECLARE @Err_Code AS INT;
    SELECT @Err_Code = 0;

    SELECT @ContactTypeID = ContactTypeID
    FROM Person.ContactType
    WHERE [Name] = @Name;

    IF @ContactTypeID IS NOT NULL
        RETURN;             -- Exit 1: if the ContactType exists

    INSERT
    INTO Person.ContactType ([Name], ModifiedDate)
    SELECT @Name, CURRENT_TIMESTAMP;

    SELECT @Err_Code = @@error;
    IF @Err_Code <> 0
        RETURN @Err_Code;   -- Exit 2: if there is an error on INSERT

    SELECT @ContactTypeID = SCOPE_IDENTITY();

    RETURN @Err_Code;       -- Exit 3: after successful INSERT
GO
```

This code has one entry point, but three possible exit points. Figure 1-3 shows a simple flowchart for the paths this code can take.

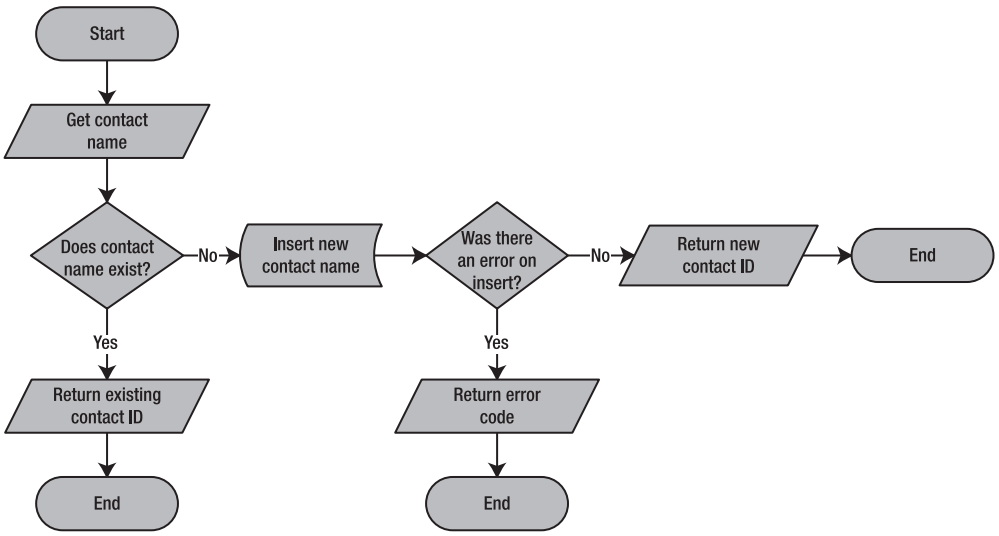

Figure 1-3. *Flowchart for example with one entry and multiple exits*

As you can imagine, maintaining code such as in Listing 1-7 becomes more difficult because the flow of the code has so many possible exit points, each of which must be accounted for when you make modifications to the SP. Listing 1-8 updates Listing 1-7 to give it a single entry point and a single exit point, making the logic easier to follow.

Listing 1-8. *Stored Procedure with One Entry and One Exit*

```
CREATE PROCEDURE dbo.GetOrAdd_ContactType
(
    @Name NVARCHAR(50),
    @ContactTypeID INT OUTPUT
)
AS
    DECLARE @Err_Code AS INT;
    SELECT @Err_Code = 0;

    SELECT @ContactTypeID = ContactTypeID
    FROM Person.ContactType
    WHERE [Name] = @Name;

    IF @ContactTypeID IS NULL
    BEGIN
        INSERT
        INTO Person.ContactType ([Name], ModifiedDate)
        SELECT @Name, CURRENT_TIMESTAMP;
```

```
        SELECT @Err_Code = @@error;
        IF @Err_Code = 0        -- If there's an error, skip next
            SELECT @ContactTypeID = SCOPE_IDENTITY();
    END
    RETURN @Err_Code;   -- Single exit point
GO
```

Figure 1-4 shows the modified flowchart for this new version of the SP.

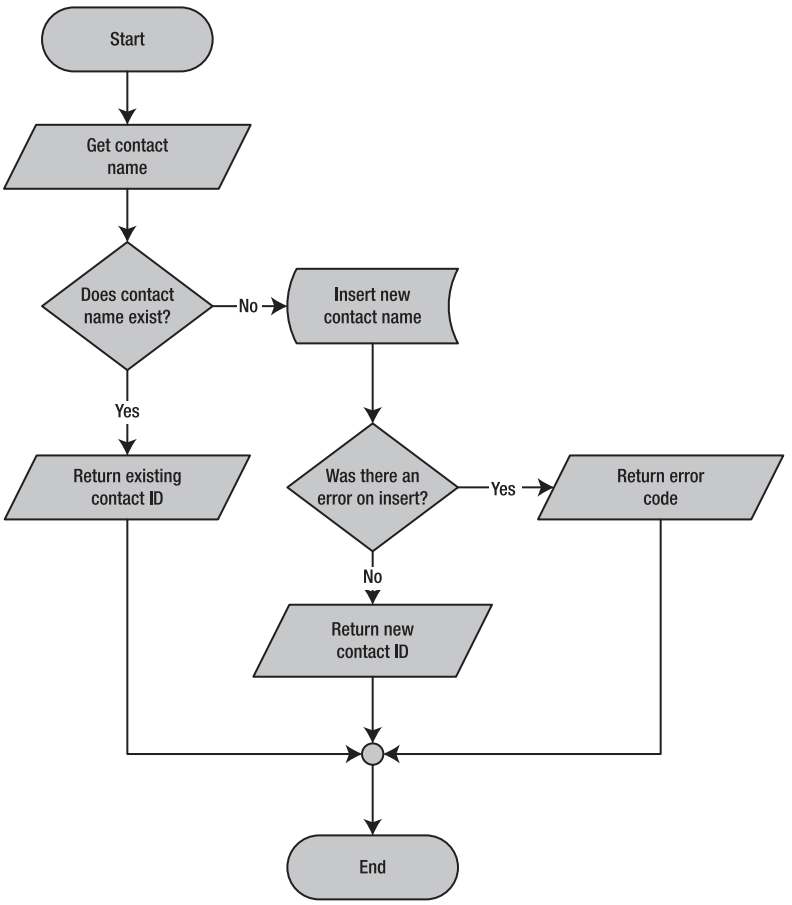

Figure 1-4. *Flowchart for example with one entry and one exit*

The single-entry/single-exit model makes the logic easier to follow, which in turn makes the code easier to manage. This rule also applies to looping structures, which you implement via the `WHILE` statement in T-SQL. Avoid using the `WHILE` loop's `CONTINUE` and `BREAK` statements and the `GOTO` statement; these statements lead to old-fashioned, difficult-to-maintain spaghetti code.

Defensive Coding

Defensive coding involves anticipating problems before they occur and mitigating them through good coding practices. The first and foremost lesson of defensive coding is to always check user input. Once you open your system up to users, expect them to do everything in their power to try to break your system. For instance, if you ask users to enter a number between 1 and 10, expect that they'll ignore your directions and key in `; DROP TABLE dbo.syscomments; --` at the first available opportunity. Defensive coding practices dictate that you should check and scrub external inputs. Don't blindly trust anything that comes from an external source.

Another aspect of defensive coding is a clear delineation between exceptions and run-of-the-mill issues. The key is that exceptions are, well, exceptional in nature. Ideally, exceptions should be caused by errors that you can't account for or couldn't reasonably anticipate, like a lost network connection or physical corruption of your application or data storage. Errors that can be reasonably expected, like data entry errors, should be captured before they are raised to the level of exceptions. Keep in mind that exceptions are often resource intensive, expensive operations. If you can avoid an exception by anticipating a particular problem, your application will benefit in both performance and control.

SQL-92 Syntax Outer Joins

Microsoft has been warning us for a long time, and starting with SQL 2005 they finally went and did it. SQL Server 2008, like SQL Server 2005, eliminates the old-style *= and =* outer join operators. Queries like the one in Listing 1-9 won't work with SQL Server 2008.

Listing 1-9. *Query Using Old-Style Join Operators*

```
SELECT o.name
FROM sys.objects o,
     sys.views v
WHERE o.object_id *= v.object_id;
```

SQL responds to this query with one of the most elaborate error messages ever seen in a Microsoft product:

```
SQL2008(SQL2008\Michael): Msg 4147, Level 15, State 1, Line 4
The query uses non-ANSI outer join operators ("*=" or "=*"). To run this query
without modification, please set the compatibility level for current database
to 80, using the SET COMPATIBILITY_LEVEL option of ALTER DATABASE. It is
strongly recommended to rewrite the query using ANSI outer join operators
(LEFT OUTER JOIN, RIGHT OUTER JOIN). In the future versions of SQL Server,
non-ANSI join operators will not be supported even in backward-compatibility
modes.
```

As the error message suggests, you can use the `sp_dbcmptlevel` SP to revert the database compatibility to `80` (SQL Server 2000) as a workaround for this problem. Best practices dictate that you should eliminate the old-style joins from your code as soon as possible. Backward-compatibility mode should be considered a temporary workaround for the old-style join issue, not a permanent fix.

COMPATIBILITY MODE CHANGES

You can use the system SP `sp_dbcmptlevel` to revert various SQL Server behaviors to a version prior to SQL Server 2000. Use a compatibility level of `90` for SQL Server 2005 and `80` for SQL Server 2000. The SP call that converts the AdventureWorks sample database to SQL Server 2000 compatibility mode looks like this:

```
EXEC sp_dbcmptlevel 'AdventureWorks', 90
```

To convert it back to SQL Server 2008 compatibility mode, you would use a statement like this:

```
EXEC sp_dbcmptlevel 'AdventureWorks', 100
```

When you set a database to backward-compatibility mode, you can lose access to some of the new functionality, such as SQL CLR support and SSMS diagrams for that database. Note that previous compatibility levels of `70` for SQL Server 7.0, `65` for SQL Server 6.5, and `60` for SQL Server 6.0 are no longer available in SQL Server 2008. Microsoft has announced that from here on out they will only be supporting two prior versions in backward-compatibility modes, so you can expect SQL Server 2000 compatibility mode to be gone with the next SQL Server release after 2008. You should avoid using backward-compatibility mode unless you have a compelling reason.

The error message also suggests that the old-style join operators will not be supported in future versions, even in backward-compatibility mode. If you have old-style joins in your T-SQL code, the best course of action is to convert them to ISO SQL standard joins as soon as possible. Listing 1-10 updates the previous query to use the current SQL standard.

Listing 1-10. *ISO SQL-92 Standard Join Syntax*

```
SELECT o.name
FROM sys.objects o
LEFT  JOIN sys.views v
    ON o.object_id = v.object_id;
```

Note that you can still use the abbreviated inner join syntax without any problems. The abbreviated inner join syntax looks like Listing 1-11.

Listing 1-11. *Abbreviated Inner Join Syntax*

```
SELECT o.name
FROM sys.objects o,
    sys.views v
WHERE o.object_id = v.object_id;
```

This book uses the ISO SQL-92 standard syntax joins exclusively in code examples.

The SELECT * Statement

Consider the `SELECT *` style of querying. In a `SELECT` clause, the asterisk (*) is a shorthand way of specifying that all columns in a table should be returned. Although `SELECT *` is a handy tool

for ad hoc querying of tables during development and debugging, you should normally not use it in a production system. One reason to avoid this method of querying is to minimize the amount of data retrieved with each call. `SELECT *` retrieves all columns, whether or not they are needed by the higher-level applications. For queries that return a large number of rows, even one or two extraneous columns can waste a lot of resources.

Also, if the underlying table or view is altered, columns might be added to or removed from the returned result set. This can cause errors that are hard to locate and fix. By specifying the column names, your front-end application can be assured that only the required columns are returned by a query, and that errors caused by missing columns will be easier to locate.

As with most things, there are always exceptions—for example, if you are using the `FOR XML AUTO` clause to generate XML based on the structure and content of your relational data. In this case, `SELECT *` can be quite useful, since you are relying on `FOR XML` to automatically generate the node names based on the table and column names in the source tables.

Variable Initialization

When you create SPs, UDFs, or any script that uses T-SQL user variables, you should initialize those variables before the first use. Unlike other programming languages that guarantee that newly declared variables will be initialized to `0` or an empty string (depending on their data types), T-SQL guarantees only that newly declared variables will be initialized to `NULL`. Consider the code snippet shown in Listing 1-12.

Listing 1-12. *Sample Code Using an Uninitialized Variable*

```
DECLARE @i INT;
SELECT @i = @i + 5;
SELECT @i;
```

The result is `NULL`, a shock if you were expecting `5`. Expecting SQL Server to initialize numeric variables to `0` (like `@i` in the previous example) or an empty string will result in bugs that can be extremely difficult to locate in your T-SQL code. To avoid these problems, always explicitly initialize your variables after declaration, as demonstrated in Listing 1-13.

Listing 1-13. *Sample Code Using an Initialized Variable*

```
DECLARE @i INT = 0; -- Changed this statement to initialize @i to 0
SELECT @i = @i + 5;
SELECT @i;
```

Summary

This chapter has served as an introduction to T-SQL, including a brief history of SQL and a discussion of the declarative programming style. I started this chapter with a discussion of ISO SQL standard compatibility in SQL Server 2008 and the differences between imperative and declarative languages, of which SQL is the latter. I also introduced many of the basic components of SQL, including databases, tables, views, SPs, and other common database objects. Finally, I provided my personal recommendations for writing SQL code that is easy to debug

and maintain. I subscribe to the "eat your own dog food" theory, and throughout this book I will faithfully follow the best practice recommendations that I've asked you to consider.

The next chapter provides an overview of the new and improved tools available out of the box for developers. Specifically, Chapter 2 will discuss the SQLCMD text-based SQL client (originally a replacement for `osql`), SSMS, SQL Server 2008 Books Online (BOL), and some of the other tools available for making writing, editing, testing, and debugging easier and faster than ever.

EXERCISES

1. Describe the difference between an imperative language and a declarative language.
2. What does the acronym *ACID* stand for?
3. SQL Server 2008 supports five different types of indexes. What are they?
4. Name two of the restrictions on any type of SQL Server UDF.
5. [True/false] In SQL Server, newly declared variables are always assigned the default value `0` for numeric data types and an empty string for character data types.