

THE EXPERT'S VOICE® IN SQL SERVER

SQL Server 2012 T-SQL Recipes

A Problem-Solution Approach

*GET THE JOB DONE WITH SQL SERVER'S
POWERFUL DATABASE PROGRAMMING AND
QUERY LANGUAGE*

Jason Brimhall, David Dye, Jonathan Gennick
Andy Roberts, and Wayne Sheffield

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.



Apress®

Contents at a Glance

About the Authors	xlix
About the Technical Reviewers	li
Acknowledgments	liii
Introduction	lv
■ Chapter 1: Getting Started with SELECT	1
■ Chapter 2: Elementary Programming	23
■ Chapter 3: NULLs and Other Pitfalls	41
■ Chapter 4: Querying from Multiple Tables	57
■ Chapter 5: Grouping and Summarizing	79
■ Chapter 6: Advanced Select Techniques	93
■ Chapter 7: Aggregations and Windowing	115
■ Chapter 8: Inserting, Updating, Deleting	147
■ Chapter 9: Working with Strings	179
■ Chapter 10: Working with Dates and Times	197
■ Chapter 11: Working with Numbers	219
■ Chapter 12: Transactions, Locking, Blocking, and Deadlocking	241
■ Chapter 13: Managing Tables	273
■ Chapter 14: Managing Views	301
■ Chapter 15: Managing Large Tables and Databases	319
■ Chapter 16: Managing Indexes	341

■ Chapter 17: Stored Procedures	363
■ Chapter 18: User-Defined Functions and Types	383
■ Chapter 19: Triggers	415
■ Chapter 20: Error Handling	447
■ Chapter 21: Query Performance Tuning	465
■ Chapter 22: Hints	507
■ Chapter 23: Index Tuning and Statistics	519
■ Chapter 24: XML	539
■ Chapter 25: Files, Filegroups, and Integrity	559
■ Chapter 26: Backup	593
■ Chapter 27: Recovery	621
■ Chapter 28: Principals and Users	639
■ Chapter 29: Securables, Permissions, and Auditing	673
■ Chapter 30: Objects and Dependencies	725
Index	737

Introduction

Sometimes all one wants is a good example. That's our motivation for accepting the baton from Joe Sack and revising his excellent work to cover the very latest edition of Microsoft's database engine—SQL Server 2012.

T-SQL is fundamental to working with SQL Server. Almost everything you do, from querying a table to creating indexes to backing up and recovering, ultimately comes down to T-SQL statements being issued and executed. Sometimes it's a utility executing statements on your behalf. Other times you must write them yourself.

And when you have to write them yourself, you're probably going to be in a hurry. Information technology is like that. It's a field full of stress and deadlines, and don't we all just want to get home for dinner with our families?

We sure do want to be home for dinner, and that brings us full circle to the example-based format you'll find in this book. If you have a job to do that's covered in this book, you can count on a clear code example and very few words to waste your time. We put the code first! And explain it afterward. We hope our examples are clear enough that you can just crib from them and get on with your day, but the detailed explanations are there if you need them.

We've missed a few dinners from working on this book. We hope it helps you avoid the same fate.

Who This Book Is For

SQL Server 2012 T-SQL Recipes is aimed at developers deploying applications against Microsoft SQL Server 2012. The book also helps database administrators responsible for managing those databases. Any developer or administrator valuing good code examples will find something of use in this book.

Conventions

Throughout the book, we've tried to keep to a consistent style for presenting SQL and results. Where a piece of code, a SQL reserved word, or a fragment of SQL is presented in the text, it is presented in fixed-width Courier font, such as this example:

```
SELECT * FROM HumanResources.Employee;
```

Where we discuss the syntax and options of SQL commands, we use a conversational style so you can quickly reach an understanding of the command or technique. We have chosen not to duplicate complex syntax diagrams that are best left to the official, vendor-supplied documentation. Instead, we take an example-based approach that is easy to understand and adapt.

Downloading the Code

The code for the examples shown in this book is available on the Apress web site, www.apress.com. A link can be found on the book's information page (www.apress.com/9781430242000) on the Source Code/Downloads tab. This tab is located in the Related Titles section of the page.

CHAPTER 1



Getting Started with SELECT

by Jonathan Gennick

The SELECT command is the cornerstone of the Transact-SQL language, allowing you to retrieve data from a SQL Server database (and more specifically from database objects within a SQL Server database). Although the full syntax of the SELECT statement is enormous, the basic syntax can be presented in a more boiled-down form:

```
SELECT select_list
FROM table_list
WHERE predicates
ORDER BY sort_key_columns;
```

The `select_list` argument is the list of columns that you wish to return in the results of the query. The `table_list` arguments are the actual tables and/or views from which the data will be retrieved. Write predicates in your WHERE clause to restrict results to rows of interest, and specify sort key columns control the ordering of results.

■ **Note** All examples in this chapter make use of the AdventureWorks database. Be sure to execute a USE AdventureWorks command to switch to that database before executing any of the examples in this chapter. If you don't already have it, you'll find the AdventureWorks example database in Microsoft's repository at www.codeplex.com. The specific URL for the SQL Server version is currently: <http://msftdbprodsamples.codeplex.com/>.

1-1. Connecting to a Database

Problem

You are running SQL Server Management Studio to execute ad hoc SQL statements. You wish to connect to a specific database, such as the example database.

Solution

Execute the USE command, and specify the name of your target database. For example, we executed the following command to attach to the example database used during work on this book:

```
USE AdventureWorks2008R2;
```

Command(s) completed successfully.

The success message indicates a successful connection. You may now execute queries against tables and views in the database without having to qualify those object names by specifying the database name each time.

How It Works

When you first launch SQL Server Management Studio you are connected by default to the master database. That's usually not convenient, and you shouldn't be storing your data in that database. You can query tables and views in other databases provided you specify fully qualified names. For example, you can specify a fully qualified name in the following, `database.schema.object` format:

```
AdventureWorks2008R2.HumanResources.Employee
```

The USE statement in the solution enables you to omit the database name and refer to the object using the shorter and simpler, `schema.object` notation. For example:

```
HumanResources.Employee
```

It's cumbersome to specify the database name—AdventureWorks2008R2 in this case—with each object reference. Doing so ties your queries and program code to a specific database, reducing flexibility by making it difficult or impossible to run against a different database in the future. Examples in this book generally assume that you are connected to the AdventureWorks example database that you can download from www.codeplex.com.

1-2. Retrieving Specific Columns

Problem

You have a table or a view. You wish to retrieve data from specific columns.

Solution

Write a SELECT statement. List the columns you wish returned following the SELECT keyword. The following example demonstrates a very simple SELECT against the AdventureWorks database, whereby three columns are returned, along with several rows from the HumanResources.Employee table.

```
SELECT NationalIDNumber,
       LoginID,
       JobTitle
FROM   HumanResources.Employee;
```

The query returns the following abridged results:

NationalIDNumber	LoginID	JobTitle
295847284	adventure-works\ken0	Chief Executive Officer
245797967	adventure-works\terri0	Vice President of Engineering
509647174	adventure-works\roberto0	Engineering Manager
112457891	adventure-works\rob0	Senior Tool Designer
695256908	adventure-works\gail0	Design Engineer
...		

How It Works

The first few lines of code define which columns to display in the query results:

```
SELECT NationalIDNumber,
       LoginID,
       JobTitle
```

The next line of code is the FROM clause:

```
FROM HumanResources.Employee;
```

The FROM clause specifies the data source, which in this example is a table. Notice the two-part name of `HumanResources.Employee`. The first part (the part before the period) is the *schema*, and the second part (after the period) is the actual table name. A schema contains the object, and that schema is then owned by a user. Because users own a schema, and the schema contains the object, you can change the owner of the schema without having to modify object ownership.

1-3. Retrieving All Columns

Problem

You are writing an ad hoc query. You wish to retrieve all columns from a table or view without having to type all the column names.

Solution

Specify an asterisk (*) instead of a column list. Doing so causes SQL Server to return all columns from the table or view. For example:

```
SELECT *
FROM HumanResources.Employee;
```

The abridged column and row output are shown here:

BusinessEntityID	NationalIDNumber	LoginID	OrganizationNode ...
1	295847284	adventure-works\ken0	0x ...
2	245797967	adventure-works\terri0	0x58 ...
3	509647174	adventure-works\roberto0	0x5AC0 ...
...			

How It Works

The asterisk symbol (*) returns all columns of the table or view you are querying. All other details are as explained in the previous recipe.

Please remember that, as good practice, it is better to reference the columns you want to retrieve explicitly instead of using `SELECT *`. If you write an application that uses `SELECT *`, your application may expect the same columns (in the same order) from the query. If later on you add a new column to the underlying table or view, or if you reorder the table columns, you could break the calling application, because the new column in your result set is unexpected.

Using `SELECT *` can also negatively affect performance, as you may be returning more data than you need over the network, increasing the result set size and data retrieval operations on the SQL Server instance. For

applications requiring thousands of transactions per second, the number of columns returned in the result set can have a nontrivial impact.

1-4. Specifying the Rows to Be Returned

Problem

You do not want to return all rows from a table or a view. You want to restrict query results to only those rows of interest.

Solution

Specify a WHERE clause giving the conditions that rows must meet in order to be returned. For example, the following query returns only rows in which the person's title is "Ms."

```
SELECT Title,
       FirstName,
       LastName
FROM   Person.Person
WHERE  Title = 'Ms.';
```

This example returns the following (abridged) results:

Title	FirstName	LastName
-----	-----	-----
Ms.	Gail	Erickson
Ms.	Janice	Galvin
Ms.	Jill	Williams
...		

You may combine multiple conditions in a WHERE clause through the use of the keywords AND and OR. The following query looks specifically for Ms. Antrim's data:

```
SELECT Title,
       FirstName,
       LastName
FROM   Person.Person
WHERE  Title = 'Ms.' AND
       LastName = 'Antrim';
```

The result from this query will be the following single row:

Title	FirstName	LastName
-----	-----	-----
Ms.	Ramona	Antrim

How It Works

In a SELECT query, the WHERE clause restricts rows returned in the query result set. The WHERE clause provides search conditions that determine the rows returned by the query. Search conditions are written as predicates,

which are expressions that evaluate to one of the Boolean results of TRUE, FALSE, or UNKNOWN. Only rows for which the final evaluation of the WHERE clause is TRUE are returned. Table 1-1 lists some of the common operators available.

Table 1-1. Operators

Operator	Description
!=	Tests two expressions not being equal to each other.
!>	Tests whether the left condition is less than or equal to (i.e., not greater than) the condition on the right.
!<	Tests whether the left condition is greater than or equal to (i.e., not less than) the condition on the right.
<	Tests the left condition as less than the right condition.
<=	Tests the left condition as less than or equal to the right condition.
<>	Tests two expressions not being equal to each other.
=	Tests equality between two expressions.
>	Tests the left condition being greater than the expression to the right.
>=	Tests the left condition being greater than or equal to the expression to the right.

■ **Tip** Don't think of a WHERE clause as going out and retrieving rows that match the conditions. Think of it as a fishnet or a sieve. All the possible rows are dropped into the net. Unwanted rows fall on through. When a query is done executing, the rows remaining in the net are those that match the predicates you listed. Database engines will optimize execution, but the fishnet metaphor is a useful one when initially crafting a query.

In this recipe's first example, you can see that only rows where the person's title was equal to "Ms." were returned. This search condition was defined in the WHERE clause of the query:

```
WHERE Title = 'Ms.'
```

You may combine multiple search conditions by utilizing the AND and OR logical operators. The AND logical operator joins two or more search conditions and returns rows only when each of the search conditions is true. The OR logical operator joins two or more search conditions and returns rows when any of the conditions are true. The second solution example shows the following AND operation:

```
WHERE Title = 'Ms.' AND
      LastName = 'Antrim'
```

Both search conditions must be true for a row to be returned in the result set. Thus, only the row for Ms. Antrim is returned.

Use the OR operator to specify alternate choices. Use parentheses to clarify the order of operations. For example:

```
WHERE Title = 'Ms.' AND
      (LastName = 'Antrim' OR LastName = 'Galvin')
```

Here, the OR expression involving the two LastName values is evaluated first, and then the Title is examined. UNKNOWN values can make their appearance when NULL data is accessed in the search condition. A NULL value doesn't mean that the value is blank or zero, only that the value is unknown. Recipe 1-7 later in this chapter shows how to identify rows having or not having NULL values.

1-5. Renaming the Output Columns

Problem

You don't like the column names returned by a query. You wish to change the names for clarity in reporting, or to be compatible with an already written program that is consuming the results from the query.

Solution

Designate column aliases. Use the AS clause for that purpose. For example:

```
SELECT BusinessEntityID AS "Employee ID",
       VacationHours AS "Vacation",
       SickLeaveHours AS "Sick Time"
FROM   HumanResources.Employee;
```

Results are as follows:

Employee ID	Vacation	Sick Time
-----	-----	-----
1	99	69
2	1	20
3	2	21
...		

How It Works

Each column in a result set is given a name. That name appears in the column heading when you execute a query ad hoc using management studio. The name is also the name by which any program code must reference the column when consuming the results from a query. You can specify any name you like for a column via the AS clause. The name you specify is termed a *column alias*.

The solution query places column names in double quotes. Follow that approach when your new name contains spaces or other nonalphanumeric or nonnumeric characters, or when you wish to specify lowercase characters and have the database configured to use an uppercase collation. For example:

```
BusinessEntityID AS "Employee ID",
```

If your new name has no spaces or other unusual characters, then you can omit the double quotes:

```
VacationHours AS Vacation,
```

You may also choose to omit the AS keyword:

```
VacationHours Vacation,
```

Well-chosen column aliases make ad hoc reports easier to comprehend. Column aliases also provide a way to insulate program code from changes in column names at the database level. They are especially helpful in that regard when you have columns that are the results of expressions. See Recipe 1-6 for an example.

SQUARE BRACKETS OR QUOTES? AS OR NOT-AS?

Recipe 1-5 shows the ISO standard syntax for handling spaces and other special characters in alias names. SQL Server also supports a proprietary syntax involving square brackets. Following are two examples that are equivalent in meaning:

```
BusinessEntityID AS "Employee ID",
BusinessEntityID AS [Employee ID],
```

Recipe 1-5 also shows that you can take or leave the AS keyword when specifying column aliases. In fact, SQL Server also supports its own proprietary syntax. Here are three examples that all mean the same thing:

```
VacationHours AS Vacation,
VacationHours Vacation,
Vacation = VacationHours,
```

I prefer to follow the ISO standard, so I write enclosing quotes whenever I must deal with unusual characters in a column alias. I also prefer the clarity of specifying the AS keyword. I avoid SQL Server's proprietary syntax in these cases.

1-6. Building a Column from an Expression

Problem

You are querying a table that lacks the precise bit of information you need. However, you are able to write an expression to generate the result that you are after. For example, you want to report on total time off available to employees. Your database design divides time off into separate buckets for vacation time and sick time. You, however, wish to report a single value.

Solution

Write the expression. Place it into the SELECT list as you would any other column. Provide a column alias by which the program executing the query can reference the column.

Following is an example showing an expression to compute the total number of hours an employee might be able to take off from work. The total includes both vacation and sick time.

```
SELECT BusinessEntityID AS EmployeeID,
       VacationHours + SickLeaveHours AS AvailableTimeOff
FROM   HumanResources.Employee;
```

EmployeeID	AvailableTimeOff
1	168
2	21
3	23
...	

How It Works

Recipe 1-5 introduces column aliases. It's especially important to provide them for computed columns. That's because if you don't provide them, you get no name at all. For example, you can omit the `AvailableTimeOff` alias as follows:

```
SELECT BusinessEntityID AS EmployeeID,
       VacationHours + SickLeaveHours
FROM   HumanResources.Employee;
```

Do so, and you'll be rewarded by a result set having a column with no name:

EmployeeID	
1	168
2	21
3	23

What is that second column? Will you remember what it is on the day after? How does program code refer to the column? Avoid these pesky questions by providing a stable column alias that you can maintain throughout the life of the query.

1-7. Providing Shorthand Names for Tables

Problem

You are writing a query and want to qualify all your column references by indicating the source table. Your table name is long. You wish for a shorter nickname by which to refer to the table.

Solution

Specify a table alias. Use the `AS` keyword to do that. For example:

```
SELECT E.BusinessEntityID AS "Employee ID",
       E.VacationHours AS "Vacation",
       E.SickLeaveHours AS "Sick Time"
FROM   HumanResources.Employee AS E;
```

How It Works

Table aliases work much like column aliases. Specify them using an `AS` clause. Place the `AS` clause immediately following the table name in your query's `FROM` clause. The solution example provides the alternate name `E` for the table `HumanResources.Employee`. As far as the rest of the query is concerned, the table is now named `E`. In fact, you may no longer refer to the table as `HumanResources.Employee`. If you try, you will get the following error:

```
SELECT HumanResources.Employee.BusinessEntityID AS "Employee ID",
       E.VacationHours AS "Vacation",
       E.SickLeaveHours AS "Sick Time"
FROM   HumanResources.Employee AS E
```

Msg 4104, Level 16, State 1, Line 1

The multi-part identifier "HumanResources.Employee.BusinessEntityID" could not be bound.

Table aliases make it much easier to fully qualify your column names in a query. It is much easier to type:

`E.BusinessEntityID`

...than it is to type:

`HumanResources.Employee.BusinessEntityID`

You may not see the full utility of table aliases now, but their benefits become readily apparent the moment you begin writing queries involving multiple tables. Chapter 4 makes extensive use of table aliases in queries involving joins and subqueries.

1-8. Negating a Search Condition

Problem

You are finding it easier to describe those rows that you do not want rather than those that you do want.

Solution

Describe the rows that you do not want. Then use the NOT operator to essentially reverse the description so that you get those rows that you do want. The NOT logical operator negates the expression that follows it.

For example, you can retrieve all employees having a title of anything but "Ms." or "Mrs." Not having yet had your morning coffee, you prefer not to think through how to translate that requirement into a conjunction of two, not-equal predicates, preferring instead to write a predicate more in line with how the problem has been described. For example:

```
SELECT Title,
       FirstName,
       LastName FROM Person.Person
WHERE NOT (Title = 'Ms.' OR Title = 'Mrs.');
```

This returns the following (abridged) results:

Title	FirstName	LastName
-----	-----	-----
Mr.	Jossef	Goldberg
Mr.	Hung-Fu	Ting
Mr.	Brian	Welcker
Mr.	Tete	Mensa-Annan
Mr.	Syed	Abbas
Mr.	Gustavo	Achong
Sr.	Humberto	Acevedo
Sra.	Pilar	Ackerman
Ms	Alyssa	Moore
...		

How It Works

This example demonstrated the NOT operator:

```
WHERE NOT (Title = 'Ms.' OR Title = 'Mrs.');
```

NOT specifies the reverse of a search condition, in this case specifying that only rows that don't have the Title equal to "Ms." or "Mrs." be returned. Rows that do represent "Ms." or "Mrs." are excluded from the results. You can also choose to write the query using a conjunction of two not-equal predicates. For example:

```
SELECT Title,
       FirstName,
       LastName FROM Person.Person
WHERE Title != 'Ms.' AND Title != 'Mrs.';
```

There is generally no right or wrong choice to be made here. Rather, your decision will most often come down to your own preference and how you tend to approach and think about query problems.

KEEPING YOUR WHERE CLAUSE UNAMBIGUOUS

You can write multiple operators (AND, OR, NOT) in a single WHERE clause, but it is important to make your intentions clear by properly embedding your ANDs and ORs in parentheses. The NOT operator takes precedence (is evaluated first) before AND. The AND operator takes precedence over the OR operator. Using both AND and OR operators in the same WHERE clause without parentheses can return unexpected results. For example, the following query may return unintended results:

```
SELECT Title,
       FirstName,
       LastName
FROM Person.Person
WHERE Title = 'Ms.' AND
       (FirstName = 'Catherine' OR
        LastName = 'Adams')
```

Is the intention to return results for all rows with a Title of "Ms.", and of those rows, only include those with a FirstName of Catherine or a LastName of Adams? Or did the query author wish to search for all people named "Ms." with a FirstName of Catherine, as well as anyone with a LastName of Adams?

It is good practice to use parentheses to clarify exactly what rows should be returned. Even if you are fully conversant with the rules of operator precedence, those who come after you may not be. Make judicious use of parentheses to remove all doubt as to your intentions.

1-9. Specifying A Range of Values

Problem

You wish to specify a range of values as a search condition. For example, you are querying a table having a date column. You wish to return rows having dates only in a specified range of interest.

Solution

Write a predicate involving the BETWEEN operator. That operator allows you to specify a range of values, in this case of date values. For example, to find sales orders placed between the dates July 23, 2005 and July 24, 2005:

```
SELECT SalesOrderID,
       ShipDate
FROM   Sales.SalesOrderHeader
WHERE  ShipDate BETWEEN '2005-07-23T00:00:00'
       AND   '2005-07-24T23:59:59';
```

The query returns the following results:

SalesOrderID	ShipDate
43758	2005-07-23 00:00:00.000
43759	2005-07-23 00:00:00.000
43760	2005-07-23 00:00:00.000
43761	2005-07-23 00:00:00.000
43762	2005-07-24 00:00:00.000
43763	2005-07-24 00:00:00.000
43764	2005-07-24 00:00:00.000
43765	2005-07-24 00:00:00.000

How It Works

This recipe demonstrates the BETWEEN operator, which tests whether a column's value falls between two values that you specify. The value range is inclusive of the two endpoints.

The WHERE clause in the solution example is written as:

```
WHERE ShipDate BETWEEN '2005-07-23T00:00:00' AND '2005-07-24T23:59:59'
```

Notice that we designate the specific time in hours, minutes, and seconds as well. The time-of-day defaults to 00:00:00, which is midnight at the start of a date. In this example, we wanted to include all of July 24, 2005. Thus we specify the last possible minute of that day.

1-10. Checking for NULL Values

Problem

Some of the values in a column might be NULL. You wish to identify rows having or not having NULL values.

Solution

Make use of the IS NULL and IS NOT NULL tests to identify rows having or not having NULL values in a given column. For example, the following query returns any rows for which the value of the product's weight is unknown:

```
SELECT ProductID,
       Name,
       Weight
FROM   Production.Product
WHERE  Weight IS NULL;
```


This query returns the following (abridged) results:

ProductID	Name	Weight
1	Adjustable Race	NULL
2	Bearing Ball	NULL
3	BB Ball Bearing	NULL
4	Headset Ball Bearings	NULL
...		

How It Works

NULL values cannot be identified using operators such as = and <> that are designed to compare two values and return a TRUE or FALSE result. NULL actually indicates the absence of a value. For that reason, neither of the following predicates can be used to detect a NULL value:

Weight = NULL yields the value UNKNOWN, which is neither TRUE nor FALSE

Weight <> NULL also yields UNKNOWN

IS NULL however, is specifically designed to return TRUE when a value is NULL. Likewise, the expression IS NOT NULL returns TRUE when a value is not NULL. Predicates involving IS NULL and IS NOT NULL enable you to filter for rows having or not having NULL values in one or more columns.

■ **Caution** NULL values and their improper handling are one of the most prevalent sources of query mistakes. See Chapter 3 for guidance and techniques that can help you avoid trouble and get the results you want.

1-11. Providing a List of Values

Problem

You are searching for matches to a specific list of values. You could write a string of predicates joined by OR operators. But you prefer a more easily readable and maintainable solution.

Solution

Create a predicate involving the IN operator, which allows you to specify an arbitrary list of values. For example, the IN operator in the following query tests the equality of the Color column to a list of expressions:

```
SELECT ProductID,
       Name,
       Color
FROM Production.Product
WHERE Color IN ('Silver', 'Black', 'Red');
```

This returns the following (abridged) results:

ProductID	Name	Color
-----	-----	-----
317	LL Crankarm	Black
318	ML Crankarm	Black
319	HL Crankarm	Black
320	Chainring Bolts	Silver
321	Chainring Nut	Silver
...		

How It Works

Use the IN operator any time you have a specific list of values. You can think of IN as shorthand for multiple OR expressions. For example, the following two WHERE clauses are semantically equivalent:

```
WHERE Color IN ('Silver', 'Black', 'Red')
```

```
WHERE Color = 'Silver' OR Color = 'Black' OR Color = 'Red'
```

You can see that an IN list becomes less cumbersome than a string of OR'd together expressions. This is especially true as the number of values grows.

■ **Tip** You can write NOT IN to find rows having values other than those that you list.

1-12. Performing Wildcard Searches

Problem

You don't have a specific value or list of values to find. What you do have is a general pattern, and you want to find all values that match that pattern.

Solution

Make use of the LIKE predicate, which provides a set of basic pattern-matching capabilities. Create a string using so-called *wildcards* to serve as a search expression. Table 1-2 shows the wildcards available in SQL Server 2012.

Table 1-2. Wildcards for the LIKE predicate

Wildcard	Usage
%	The percent sign. Represents a string of zero or more characters
_	The underscore. Represents a single character
[...]	A list of characters enclosed within square brackets. Represents a single character from among any in the list. You may use the hyphen as a shorthand to translate a range into a list. For example, [ABCDEF]-flat can be written more succinctly as [A-F]-flat. You can also mix and match single characters and ranges. The expressions [A-CD-F]-flat, [A-DEF]-flat, and [ABC-F]-flat all mean the same thing and ultimately resolve to [ABCDEF]-flat.
[^...]	A list of characters enclosed within square brackets and preceded by a caret. Represents a single character from among any <i>not</i> in the list.

The following example demonstrates using the LIKE operation with the % wildcard, searching for any product with a name beginning with the letter B:

```
SELECT ProductID,
       Name
FROM   Production.Product
WHERE  Name LIKE 'B%';
```

This query returns the following results:

ProductID	Name
3	BB Ball Bearing
2	Bearing Ball
877	Bike Wash - Dissolver
316	Blade

What if you want to search for the literal % (percentage sign) or an _ (underscore) in your character column? For this, you can use an ESCAPE operator. The ESCAPE operator allows you to search for a wildcard symbol as an actual character. First modify a row in the Production.ProductDescription table, adding a percentage sign to the Description column:

```
UPDATE Production.ProductDescription
SET   Description = 'Chromoly steel. High % of defects'
WHERE ProductDescriptionID = 3;
```

Next, query the table, searching for any descriptions containing the literal percentage sign:

```
SELECT ProductDescriptionID,
       Description
FROM   Production.ProductDescription
WHERE  Description LIKE '%/%%' ESCAPE '/';
```

Notice the use of /% in the middle of the search string passed to LIKE. The / is the ESCAPE operator. Thus, the characters /% are interpreted as %, and the LIKE predicate will identify strings containing a % in any position. The query given will return the following row:

ProductDescriptionID	Description
3	Chromoly steel. High % of defects

How It Works

Wildcards allow you to search for patterns in character-based columns. In the example from this recipe, the % percentage sign represents a string of zero or more characters:

```
WHERE Name LIKE 'B%'
```

If searching for a literal that would otherwise be interpreted by SQL Server as a wildcard, you can use the ESCAPE clause. The example from this recipe searches for a literal percentage sign in the Description column:

```
WHERE Description LIKE '%/%%' ESCAPE '/'
```

A slash embedded in single quotes was put after the ESCAPE command. This designates the slash symbol as the escape character for the preceding LIKE expression string. Any wildcard preceded by a slash is then treated as just a regular character.

1-13. Sorting Your Results

Problem

You are executing a query, and you wish the results to come back in a specific order.

Solution

Write an ORDER BY clause into your query. Specify the columns on which to sort. Place the clause at the very end of your query.

This next example demonstrates ordering the query results by columns ProductID and EndDate:

```
SELECT p.Name,
       h.EndDate,
       h.ListPrice
FROM   Production.Product AS p
       INNER JOIN Production.ProductListPriceHistory AS h
         ON p.ProductID = h.ProductID
ORDER BY p.Name,
         h.EndDate;
```

This query returns results as follows:

Name	EndDate	ListPrice
All-Purpose Bike Stand	NULL	159.00
AWC Logo Cap	NULL	8.99
AWC Logo Cap	2006-06-30 00:00:00.000	8.6442
AWC Logo Cap	2007-06-30 00:00:00.000	8.6442
Bike Wash - Dissolver	NULL	7.95
Cable Lock	2007-06-30 00:00:00.000	25.00
...		

Notice the results are first sorted on Name. Within Name, they are sorted on EndDate.

How It Works

Although queries sometimes appear to return data properly without an ORDER BY clause, you should never depend upon any ordering that is accidental. You must write an ORDER BY into your query if the order of the result

set is critical. You can designate one or more columns in your `ORDER BY` clause, as long as the columns do not exceed 8,060 bytes in total.

■ **Caution** We can't stress enough the importance of `ORDER BY` when order matters. Grouping operations and indexing sometimes make it seem that `ORDER BY` is superfluous. It isn't. Trust us: there are enough corner cases that sooner or later you'll be caught out. If the sort order matters, then say so explicitly in your query by writing an `ORDER BY` clause.

In the solution example, the `Production.Product` and `Production.ProductListPriceHistory` tables are queried to view the history of product prices over time. The query involves an inner join, and there is more about those in Chapter 4. The following line of code sorted the results first alphabetically by product name, and then by the end date:

```
ORDER BY p.Name, h.EndDate
```

The default sort order is an ascending sort. `NULL` values sort to the top in an ascending sort.

■ **Note** Need a descending sort? No problem. Just drop into the next recipe for an example.

1-14. Specifying Sort Order

Problem

You do not want the default, ascending-order sort. You want to sort by one or more columns in descending order.

Solution

Make use of the keywords `ASC` and `ASCENDING`, or `DESC` and `DESCENDING`, to specify the sort direction. Apply these keywords to each sort column as you desire.

This next example sorts on the same two columns as Recipe 1-13's query, but this time in descending order for each of those columns:

```
SELECT p.Name,
       h.EndDate,
       h.ListPrice
FROM   Production.Product AS p
       INNER JOIN Production.ProductListPriceHistory AS h
         ON p.ProductID = h.ProductID
ORDER BY p.Name DESC,
        h.EndDate DESC;
```

Following are some of the results:

Name	EndDate	ListPrice
-----	-----	-----
Women's Tights, S	2007-06-30 00:00:00.000	74.99
Women's Tights, M	2007-06-30 00:00:00.000	74.99

Women's Tights, L	2007-06-30 00:00:00.000	74.99
...		
Sport-100 Helmet, Red	2007-06-30 00:00:00.000	33.6442
Sport-100 Helmet, Red	2006-06-30 00:00:00.000	33.6442
Sport-100 Helmet, Red	NULL	34.99
...		

How It Works

Use the keywords `ASC` and `DESC` on a column-by-column basis to specify whether you want an ascending or descending sort on that column's values. If you prefer it, you can spell out the words as `ASCENDING` and `DESCENDING`.

You need not specify the same sort order for all columns listed in the `ORDER BY` clause. How each column's values are sorted is independent of the other columns. It is perfectly reasonable, for example, to specify an ascending sort by product name and a descending sort by end date.

`NULL` values in a descending sort are sorted to the bottom. You can see that in the solution results. The `NULL` value for the Sport-100 Helmet's end date is at the end of the list for that helmet.

1-15. Sorting by Columns Not Selected

Problem

You want to sort by columns not returned by the query.

Solution

Simply specify the columns you wish to sort by. They do not need to be in your query results. For example, you can return a list of product names sorted by color without returning the colors:

```
SELECT p.Name
FROM Production.Product AS p
ORDER BY p.Color;
```

Results from this query are:

```
Name
-----
Guide Pulley
LL Grip Tape
ML Grip Tape
HL Grip Tape
Thin-Jam Hex Nut 9
Thin-Jam Hex Nut 10
...
```

How It Works

You can sort by any column. It doesn't matter whether that column is in the SELECT list. What does matter is that the column must be available to the query. The solution query is against the Product table. Color is a column in that table, so it is available as a sort key.

One caveat when ordering by unselected columns is that ORDER BY items must appear in the SELECT list if SELECT DISTINCT is specified. That's because the grouping operation used internally to eliminate duplicate rows from the result set has the effect of disassociating rows in the result set from their original underlying rows in the table. That behavior makes perfect sense when you think about it. A deduplicated row in a result set would come from what originally were two or more table rows. And which of those rows would you go to for the excluded column? There is no answer to that question, and hence the caveat.

1-16. Forcing Unusual Sort Orders

Problem

You wish to force a sort order not directly supported by the data. For example, you wish to retrieve only the colored products, and you further wish to force the color red to sort first.

Solution

Write an expression to translate values in the data to values that will give the sort order you are after. Then order your query results by that expression. Following is one approach to the problem of retrieving colored parts and listing the red ones first:

```
SELECT p.ProductID,
       p.Name,
       p.Color
FROM   Production.Product AS p
WHERE  p.Color IS NOT NULL
ORDER BY CASE p.Color
          WHEN 'Red' THEN NULL
          ELSE p.Color
        END;
```

Results will be as follows:

ProductID	Name	Color
706	HL Road Frame - Red, 58	Red
707	Sport-100 Helmet, Red	Red
725	LL Road Frame - Red, 44	Red
726	LL Road Frame - Red, 48	Red
...		
790	Road-250 Red, 48	Red
791	Road-250 Red, 52	Red
792	Road-250 Red, 58	Red
793	Road-250 Black, 44	Black
794	Road-250 Black, 48	Black
...		

How It Works

The solution takes advantage of the fact that SQL Server sorts nulls first. The CASE expression returns NULL for red-colored items, thus forcing those first. Other colors are returned unchanged. The result is all the red items first in the list, and then red is followed by other colors in their natural sort order.

You don't have to rely upon nulls sorting first. Here is another version of the query to illustrate that and one other point:

```
SELECT p.ProductID,
       p.Name,
       p.Color
FROM   Production.Product AS p
WHERE  p.Color IS NOT NULL
ORDER BY CASE LOWER(p.Color)
          WHEN 'red' THEN ' '
          ELSE LOWER(p.Color)
END;
```

This version of the query returns the same results as before. The value 'Red' is converted into a single space, which sorts before all the spelled-out color names. The CASE expression specifies LOWER(p.CoLor) to ensure 'Red', 'RED', 'red', and so forth are all treated the same. Other color values are forced to lowercase to prevent any case-sensitivity problems in the sort.

1-17. Paging Through A Result Set Problem

You wish to present a result set to an application user N rows at a time.

Solution

Make use of the query paging feature that is brand new in SQL Server 2012. Do this by adding OFFSET and FETCH clauses to your query's ORDER BY clause. For example, the following query uses OFFSET and FETCH to retrieve the first 10 rows of results:

```
SELECT ProductID, Name
FROM   Production.Product
ORDER BY Name
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

Results from this query will be the first 10 rows, as ordered by product name:

ProductID	Name
1	Adjustable Race
879	All-Purpose Bike Stand
712	AWC Logo Cap
3	BB Ball Bearing
2	Bearing Ball
877	Bike Wash - Dissolver
316	Blade

843	Cable Lock
952	Chain
324	Chain Stays

Changing the offset from 0 to 8 will fetch another 10 rows. The offset will skip the first eight rows. There will be a two-row overlap with the preceding result set. Here is the query:

```
SELECT ProductID, Name
FROM Production.Product
ORDER BY Name
OFFSET 8 ROWS FETCH NEXT 10 ROWS ONLY;
```

And here are the results:

ProductID	Name
-----	-----
952	Chain
324	Chain Stays
322	Chainring
320	Chainring Bolts
321	Chainring Nut
866	Classic Vest, L
865	Classic Vest, M
864	Classic Vest, S
505	Cone-Shaped Race
323	Crown Race

Continue modifying the offset each time, paging through the result until the user is finished.

How It Works

OFFSET and FETCH turn a SELECT statement into a query fetching a specific window of rows from those possible. Use OFFSET to specify how many rows to skip from the beginning of the possible result set. Use FETCH to set the number of rows to return. You can change either value as you wish from one execution to the next.

You must specify an ORDER BY clause! OFFSET and FETCH are actually considered as part of that clause. If you don't specify a sort order, then rows can come back in any order. What does it mean to ask for the second set of 10 rows returned in random order? It doesn't really mean anything.

Be sure to specify a deterministic set of sort columns in your ORDER BY clause. Each SELECT to get the next page of results is a separate query and a separate sort operation. Make sure that your data sorts the same way each time. Do not leave ambiguity.

■ **Note** The word *deterministic* means that the same inputs always give the same outputs. Specify your sort such that the same set of input rows will always yield the same ordering in the query output.

Each execution of a paging query is a separate execution from the others. Consider executing sequences of paging queries from within a transaction providing a snapshot or serializable isolation. Chapter 12 discusses transactions in detail. However, you can begin and end such a transaction as follows:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;  
BEGIN TRANSACTION;  
... /* Queries go here */  
COMMIT;
```

Anomalies are possible without isolation. For example:

- You might see a row twice. In the solution example, if another user inserted eight new rows with names sorting earlier than “Adjustable Race,” then the second query results would be the same as the first.
- You might miss rows. If another user quickly deleted the first eight rows, then the second solution query would miss everything from “Chainring” to “Crown Race.”

You may decide to risk the default isolation level. If your target table is read-only, or if it is updated in batch-mode only at night, then you might be justified in leaving the isolation level at its default because the risk of change during the day is low to non-existent. Possibly you might choose not to worry about the issue at all. However, make sure that whatever you do is the result of thinking things through and making a conscious choice.

■ **Note** It may seem rash for us to even hint at not allowing the possibility of inconsistent results. We advocate making careful and conscious decisions. Some applications—Facebook is a well-known example—trade away some consistency in favor of performance. (We routinely see minor inconsistencies on our Facebook walls.) We are not saying you should do the same. We simply acknowledge the possibility of such a choice.



Elementary Programming

by Jonathan Gennick

In this chapter, you'll find recipes showing several of the basic programming constructs available in T-SQL. The chapter is not a complete tutorial to the language. You'll need to read other books for that. A good tutorial, if you need one that begins with first-principles, is *Beginning T-SQL 2012* by Scott Shaw and Kathi Kellenberger (Apress, 2012). What you will find in this chapter, though, are fast examples of commonly used constructs such as IF and CASE statements, WHILE loops, and T-SQL cursors.

2-1. Declaring Variables

Problem

You want to declare a variable and use it in subsequent T-SQL statements. For example, you want to build a search string, store that search string into a variable, and reference the string in the WHERE clause of a subsequent query.

Solution

Execute a DECLARE statement. Specify the variable and the data type. Optionally provide an initial value.

The following example demonstrates using a variable to hold a search string. The variable is declared and initialized to a value. Then a SELECT statement finds people with names that include the given string.

```
DECLARE @AddressLine1 nvarchar(60) = 'Heiderplatz';
SELECT AddressID, AddressLine1
FROM Person.Address
WHERE AddressLine1 LIKE '%' + @AddressLine1 + '%';
```

The query in this example returns all rows with an address containing the search string value.

AddressID	AddressLine1
20333	Heiderplatz 268
17062	Heiderplatz 268
24962	Heiderplatz 662
15742	Heiderplatz 662
27109	Heiderplatz 772
23496	Heiderplatz 772
...	

How It Works

Throughout the book you'll see examples of variables being used within queries and module-based SQL Server objects (stored procedures, triggers, and more). Variables are objects you can create to temporarily contain data. Variables can be defined across several different data types and then referenced within the allowable context of that type.

The solution query begins by declaring a new variable that is prefixed by the @ symbol and followed by the defining data type that will be used to contain the search string. Here's an example:

```
DECLARE @AddressLine1 nvarchar(60)
```

Next and last in the declaration is the initial value of the variable:

```
DECLARE @AddressLine1 nvarchar(60) = 'Heiderplatz';
```

You can also specify a value by executing a SET statement, and prior to SQL Server 2008, you are *required* to do so. Here's an example:

```
DECLARE @AddressLine1 nvarchar(60);
SET @AddressLine1 = 'Heiderplatz';
```

Next the solution executes a query referencing the variable in the WHERE clause, embedding it between the % wildcards to find any row with an address containing the search string:

```
WHERE AddressLine1 LIKE '%' + @AddressLine1 + '%'
```

It's possible to declare a variable without assigning a value. In that case, the variable is said to be *null*. Here's an example:

```
DECLARE @AddressLine1 nvarchar(60);
SELECT @AddressLine1;
Results from this query are as follows:
```

```
-----
NULL
(1 row(s) affected)
```

It is the same with a variable as with a table column. A null column is one having no value. Likewise, a null variable is one having no value.

2-2. Retrieving a Value into a Variable

Problem

You want to retrieve a value from the database into a variable for use in later T-SQL code.

Solution

Issue a query that returns zero or one rows. Specify the primary key, or a unique key, of the target row in your WHERE clause. Assign the column value to the variable, as shown in the following example:

```
DECLARE @AddressLine1 nvarchar(60);
DECLARE @AddressLine2 nvarchar(60);
SELECT @AddressLine1 = AddressLine1, @AddressLine2 = AddressLine2
FROM Person.Address
WHERE AddressID = 66;
SELECT @AddressLine1 AS Address1, @AddressLine2 AS Address2;
```

The results are as follows:

Address1	Address2
-----	-----
4775 Kentucky	Dr. Unit E

How It Works

The solution query retrieves the two address lines for address #66. Because `AddressID` is the table's primary key, there can be only one row with ID #66. A query such as in the example that can return at most one row is sometimes termed a *singleton select*.

■ **Caution** It is critical when using the technique in this recipe to make sure to write queries that can return at most one row. Do that by specifying either a primary key or a unique key in the `WHERE` clause.

The key syntax aspect to focus on is the following pattern in the `SELECT` list for assigning values returned by the query to variables that you declare:

```
@VariableName = ColumnName
```

The solution query contains two such assignments: `@AddressLine1 = AddressLine1` and `@AddressLine2 = AddressLine2`. They assign the values from the columns `AddressLine1` and `AddressLine2`, respectively, into the variables `@AddressLine1` and `@AddressLine2`.

What if your query returns no rows? In that case, your target variables will be left unchanged. For example, execute the following query block:

```
DECLARE @AddressLine1 nvarchar(60) = '101 E. Varnum'
DECLARE @AddressLine2 nvarchar(60) = 'Ambulance Desk'
SELECT @AddressLine1 = AddressLine1, @AddressLine2 = AddressLine2
FROM Person.Address
WHERE AddressID = 49862;
SELECT @AddressLine1, @AddressLine2;
```

You will get the following results:

```
-----
```

101 E. Varnum	Ambulance Desk
---------------	----------------

Now you have a problem. How do you know whether the values in the variables are from the query or whether they are left over from prior code? One solution is to test the global variable `@@ROWCOUNT`. Here's an example:

```
DECLARE @AddressLine1 nvarchar(60) = '101 E. Varnum'
DECLARE @AddressLine2 nvarchar(60) = 'Ambulance Desk'
SELECT @AddressLine1 = AddressLine1, @AddressLine2 = AddressLine2
FROM Person.Address
WHERE AddressID = 49862;
IF @@ROWCOUNT = 1
    SELECT @AddressLine1, @AddressLine2
```

ELSE

```
SELECT 'Either no rows or too many rows found.';
```

If @@ROWCOUNT is 1, then our singleton select is successful. Any other value indicates a problem. A @@ROWCOUNT of zero indicates that no row was found. A @@ROWCOUNT greater than zero indicates that more than one row was found. If multiple rows are found, you will arbitrarily be given the values from the last row in the result set. That is rarely desirable behavior and is the reason for our strong admonition to query by either the primary key or a unique key.

2-3. Writing an IF...THEN...ELSE Statement

Problem

You want to write an IF...THEN...ELSE statement so that you can control which of two possible code paths is taken.

Solution

Write your statement using the following syntax:

```
IF Boolean_expression
{ sql_statement | statement_block }
[ ELSE
{ sql_statement | statement_block } ]
```

For example, the following code block demonstrates executing a query conditionally based on the value of a local variable:

```
DECLARE @QuerySelector int = 3;
IF @QuerySelector = 1
BEGIN
    SELECT TOP 3 ProductID, Name, Color
    FROM Production.Product
    WHERE Color = 'Silver'
    ORDER BY Name
END
ELSE
BEGIN
    SELECT TOP 3 ProductID, Name, Color
    FROM Production.Product
    WHERE Color = 'Black'
    ORDER BY Name
END;
```

This code block returns the following results:

ProductID	Name	Color
322	Chainring	Black
863	Full-Finger Gloves, L	Black
862	Full-Finger Gloves, M	Black