**Chap te r 2**

Getting Started with the

SELECT Statement

Exam objectives in this chapter:

■■ Work with Data

■■ Query data by using SELECT statements.

■■ Implement data types.

■■ Modify Data

■■ Work with functions.

The previous chapter provided you with the foundations to T-SQL. This chapter starts

by covering two of the principal query clauses—FROM and SELECT. It then continues

by covering the data types supported by Microsoft SQL Server and the considerations in

choosing the appropriate data types for your columns. This chapter also covers the use of

built-in scalar functions, the CASE expression, and variations like ISNULL and COALESCE.

Lessons in this chapter:

■■ Lesson 1: Using the FROM and SELECT Clauses

■■ Lesson 2: Working with Data Types and Built-in Functions

**Before You Begin**

To complete the lessons in this chapter, you must have:

■■ Experience working with SQL Server Management Studio (SSMS).

■■ Some experience writing T-SQL code.

■■ Access to a SQL Server 2012 instance with the sample database TSQL2012 installed.

**Lesson 1: Using the FR OM and SE LECT Clauses**

The FROM and SELECT clauses are two principal clauses that appear in almost every query

that retrieves data. This lesson explains the purpose of these clauses, how to use them, and

best practices associated with them.

After this lesson, you will be able to:

■■ Write queries that use the FROM and SELECT clauses.

■■ Define table and column aliases.

■■ Describe best practices associated with the FROM and SELECT clauses.

Estimated lesson time: 30 minutes

The FROM Clause

According to logical query processing (see details in Chapter 1, “Foundations of Querying,”

explaining the concept), the FROM clause is the first clause to be evaluated logically in a

SELECT query. The FROM clause has two main roles:

■■ It’s the clause where you indicate the tables that you want to query.

■■ It’s the clause where you can apply table operators like joins to input tables.

This chapter focuses on the first role. Chapter 4, “Combining Sets,” and Chapter 5, “Grouping

and Windowing,” cover the use of table operators.

As a basic example, assuming you are connected to the sample database TSQL2012, the following

query uses the FROM clause to specify that HR.Employees is the table being queried.

SELECT empid, firstname, lastname

FROM HR.Employees;

Observe the use of the two-part name to refer to the table. The first part (HR) is the

schema name and the second part (Employees) is the table name. In some cases, T-SQL supports

omitting the schema name, as in FROM Employees, in which case it uses an implicit

schema name resolution process. It is considered a best practice to always explicitly indicate

the schema name. This practice can prevent you from ending up with a schema name that

you did not intend to be used, and can also remove the cost involved in the implicit resolution

process, although this cost is minor.

In the FROM clause, you can alias the queried tables with your chosen names. You can use

the form <*table*> <*alias*>, as in HR.Employees E, or <*table*> AS <*alias*>, as in HR.Employees

AS E. The latter form is more readable. When using aliases, the convention is to use short

names, typically one letter that is somehow indicative of the queried table, like E for Employees.

The reasons why you might want to alias tables become apparent in Chapter 4. For now,

it’s sufficient for you to know that the language supports such table aliases and the syntax to

assign them.

Note that if you assign an alias to a table, you basically rename the table for the duration

of the query. The original table name isn’t visible anymore; only the alias is. Normally, you can

prefix a column name you refer to in a query with the table name, as in Employees.empid.

However, if you aliased the Employees table as E, the reference Employees.empid is invalid;

you have to use E.empid, as the following example demonstrates.

SELECT E.empid, firstname, lastname

FROM HR.Employees AS E;

If you try running this code by using the full table name as the column prefix, the code

will fail.

As mentioned, Chapter 4 gets into the details of why table aliasing is needed.

The SELECT Clause

The SELECT clause of a query has two main roles:

■■ It evaluates expressions that define the attributes in the query’s result, assigning them

with aliases if needed.

■■ Using a DISTINCT clause, you can eliminate duplicate rows in the result if needed.

I’ll start with the first role. Take the following query as an example.

SELECT empid, firstname, lastname

FROM HR.Employees;

The FROM clause indicates that the HR.Employees table is the input table of the query.

The SELECT clause then projects only three of the attributes from the input as the returned

attributes in the result of the query.

T-SQL supports using an asterisk (\*) as an alternative to listing all attributes from the input

tables, but this is considered a bad practice for a number of reasons. Often, you need to return

only a subset of the input attributes, and using an \* is just a matter of laziness. By returning

more attributes than you really need, you can prevent SQL Server from using what would

normally be considered covering indexes in respect to the interesting set of attributes. You

also send more data than is needed over the network, and this can have a negative impact on

the system’s performance. In addition, the underlying table definition could change over time;

even if, when the query was initially authored, \* really represented all attributes you needed;

it might not be the case anymore at a later point in time. For these reasons and others, it is

considered a best practice to always explicitly list the attributes that you need.

In the SELECT clause, you can assign your own aliases to the expressions that define the

result attributes. There are a number of supported forms of aliasing: <*expression*> AS <*alias*>

as in empid AS employeeid, <*expression*> <*alias*> as in empid employeeid, and <*alias*> =

<*expression*> as in employeeid = empid.

*REAL WORLD* **A Preferred Method**

We prefer to use the first form with the AS clause because it’s both standard and we find it

to be the most readable. The second form is both unreadable and makes it hard to spot a

certain bug in the code.

Consider the following query.

SELECT empid, firstname lastname

FROM HR.Employees;

The developer who authored the query intended to return the attributes empid, firstname,

and lastname but missed indicating the comma between firstname and lastname. The

query doesn’t fail; instead, it returns the following result.

Empid lastname

----------- ----------

1 Sara

2 Don

3 Judy

...

Although not the author’s intention, SQL Server interprets the request as assigning the

alias lastname to the attribute firstname instead of returning both. If you’re used to aliasing

expressions with the space form as a common practice, it will be harder for you to spot

such bugs.

Back to intentional attribute aliasing, there are two main uses for those. One is renaming—

when you need the result attribute to be named differently than the source attribute—for

example, if you need to name the result attribute employeeid instead of empid, as follows.

SELECT empid AS employeeid, firstname, lastname

FROM HR.Employees;

Another use is to assign a name to an attribute that results from an expression that would

otherwise be unnamed. For example, suppose you need to generate a result attribute from an

expression that concatenates the firstname attribute, a space, and the lastname attribute. You

use the following query.

SELECT empid, firstname + N' ' + lastname

FROM HR.Employees;

You get a nonrelational result because the result attribute has no name.

empid

----------- -------------------------------

1 Sara Davis

2 Don Funk

3 Judy Lew

...

By aliasing the expression, you assign a name to the result attribute, making the result of

the query relational, as follows.

SELECT empid, firstname + N' ' + lastname AS fullname

FROM HR.Employees;

Here’s an abbreviated form of the result of this query.

empid fullname

----------- -------------------------------

1 Sara Davis

2 Don Funk

3 Judy Lew

...

Remember from the discussions in Chapter 1 that if duplicates are possible in the result,

T-SQL won’t try to eliminate those unless instructed. A result with duplicates is considered

nonrelational because relations—being sets—are not supposed to have duplicates. Therefore,

if duplicates are possible in the result, and you want to eliminate them in order to return a

relational result, you can do so by adding a DISTINCT clause, as in the following.

SELECT DISTINCT country, region, city

FROM HR.Employees;

The HR.Employees table has nine rows but five distinct locations; hence, the output of this

query has five rows.

country region city

--------------- --------------- ---------------

UK NULL London

USA WA Kirkland

USA WA Redmond

USA WA Seattle

USA WA Tacoma

There’s an interesting difference between standard SQL and T-SQL in terms of minimal

SELECT query requirements. According to standard SQL, a SELECT query must have at

minimum FROM and SELECT clauses. Conversely, T-SQL supports a SELECT query with only a

SELECT clause and without a FROM clause. Such a query is as if issued against an imaginary

table that has only one row. For example, the following query is invalid according to standard

SQL but is valid according to T-SQL.

SELECT 10 AS col1, 'ABC' AS col2;

The output of this query is a single row with attributes resulting from the expressions with

names assigned using the aliases.

col1 col2

----------- ----

10 ABC

Delimiting Identifiers

When referring to identifiers of attributes, schemas, tables, and other objects, there are cases

in which you are required to use delimiters vs. cases in which the use of delimiters is optional.

T-SQL supports both a standard form to delimit identifiers using double quotation marks, as

in "Sales"."Orders", as well as a proprietary form using square brackets, as in [Sales].[Orders].

When the identifier is “regular,” delimiting it is optional. In a regular identifier, the identifier

complies with the rules for formatting identifiers. The rules say that the first character must be

a letter in the range A through Z (lower or uppercase), underscore (\_), at sign (@), or number

sign (#). Subsequent characters can include letters, decimal numbers, at sign, dollar sign ($),

number sign, or underscore. The identifier cannot be a reserved keyword in T-SQL, cannot

have embedded spaces, and must not include supplementary characters.

An identifier that doesn’t comply with these rules must be delimited. For example, an

attribute called 2006 is considered an irregular identifier because it starts with a digit, and

therefore must be delimited as "2006" or [2006]. A regular identifier such as y2006 can be

referenced without delimiters simply as y2006, or it can be optional with delimiters. You

might prefer not to delimit regular identifiers because the delimiters tend to clutter the code.

**Quick Check**

1. What are the forms of aliasing an attribute in T-SQL?

2. What is an irregular identifier?

**Quick Check Answer**

1. The forms are <*expression*> AS <*alias*>, <*expression*> <*alias*>, and

<*alias*> = <*expression*>.

2. An identifier that does not follow the rules for formatting identifiers; for

example, it starts with a digit, has an embedded space, or is a reserved T-SQL

keyword.

Practice **Using the FR OM and SE LECT Clauses**

In this practice, you exercise your knowledge of using the FROM and SELECT clauses.

If you encounter a problem completing an exercise, you can install the completed projects

from the Solution folder that is provided with the companion content for this chapter and

lesson.

Exercise 1 Write a Simple Query and Use Table Aliases

In this exercise, you practice the use of the FROM and SELECT clauses, including the use of

table aliases.

1. Open SSMS and connect to the sample database TSQL2012.

2. To practice writing a simple query that uses the FROM and SELECT clauses, type the

following query and execute it.

USE TSQL2012;

SELECT shipperid, companyname, phone

FROM Sales.Shippers;

The USE statement ensures that you are connected to the target database TSQL2012.

The FROM clause indicates that the Sales.Shippers table is the queried table, and the

SELECT clause projects the attributes shipperid, companyname, and phone from this

table. Here’s the result of the query.

shipperid companyname phone

---------- -------------- ---------------

1 Shipper GVSUA (503) 555-0137

2 Shipper ETYNR (425) 555-0136

3 Shipper ZHISN (415) 555-0138

3. If there was more than one table involved in the query and another table had an attribute

called shipperid, you would need to prefix the shipperid attribute with the table

name, as in Shippers.shipperid. For brevity, you can alias the table with a shorter name,

like S, and then refer to the attribute as S.shipperid. Here’s an example for aliasing the

table and prefixing the attribute with the new table name.

SELECT S.shipperid, companyname, phone

FROM Sales.Shippers AS S;

Exercise 2 Use Column Aliases and Delimited Identifiers

In this exercise, you practice the use of column aliases, including the use of delimited identifiers.

As your starting point, you use the query from step 3 in the previous exercise.

1. Suppose you want to rename the result attribute phone to **phone number**. Here’s an

attempt to alias the attribute with the identifier phone number without delimiters.

SELECT S.shipperid, companyname, phone AS phone number

FROM Sales.Shippers AS S;

2. This code fails because phone number is not a regular identifier, and therefore has to

be delimited, as follows.

SELECT S.shipperid, companyname, phone AS [phone number]

FROM Sales.Shippers AS S;

3. Remember that T-SQL supports both a proprietary way to delimit identifiers by using

square brackets, and the standard form using double quotation marks, as in "phone

number".

Lesson Summary

■■ The FROM clause is the first clause to be logically processed in a SELECT query. In this

clause, you indicate the tables you want to query and table operators. You can alias

tables in the FROM clause with your chosen names and then use the table alias as a

prefix to attribute names.

■■ With the SELECT clause, you can indicate expressions that define the result attributes.

You can assign your own aliases to the result attributes, and in this way, create a relational

result. If duplicates are possible in the result, you can eliminate those by specifying

the DISTINCT clause.

■■ If you use regular identifiers as object and attribute names, using delimiters is optional.

If you use irregular identifiers, delimiters are required.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You

can find the answers to these questions and explanations of why each answer choice is correct

or incorrect in the “Answers” section at the end of this chapter.

1. What is the importance of the ability to assign attribute aliases in T-SQL? (Choose all

that apply.)

A. The ability to assign attribute aliases is just an aesthetic feature.

B. An expression that is based on a computation results in no attribute name unless

you assign one with an alias, and this is not relational.

C. T-SQL requires all result attributes of a query to have names.

D. Using attribute aliases, you can assign your own name to a result attribute if you

need it to be different than the source attribute name.

2. What are the mandatory clauses in a SELECT query, according to T-SQL?

A. The FROM and SELECT clauses

B. The SELECT and WHERE clauses

C. The SELECT clause

D. The FROM and WHERE clauses

3. Which of the following practices are considered bad practices? (Choose all that apply.)

A. Aliasing columns by using the AS clause

B. Aliasing tables by using the AS clause

C. Not assigning column aliases when the column is a result of a computation

D. Using \* in the SELECT list

**Lesson 2: Working with Data Types and**

**Built-in Functions**

When defining columns in tables, parameters in procedures and functions, and variables in

T-SQL batches, you need to choose a data type for those. The data type constrains the data

that is supported, in addition to encapsulating behavior that operates on the data, exposing it

through operators and other means. Because data types are such a fundamental component

of your data—everything is built on top—your choices of data types will have dramatic implications

for your application at many different layers. Therefore, this is an area that should

not be taken lightly, but instead treated with a lot of care and attention. That’s also the reason

why this topic is covered so early in this Training Kit, even though the first few chapters of the

kit focus on querying, and only later chapters deal with data definition, like creating and altering

tables. Your knowledge of types is critical for both data definition and data manipulation.

T-SQL supports many built-in functions that you can use to manipulate data. Because

functions operate on input values and return output values, an understanding of built-in

functions goes hand in hand with an understanding of data types.

Note that this chapter is not meant to be an exhaustive coverage of all types and all functions

that T-SQL supports—this would require a whole book in its own right. Instead, this chapter

explains the factors you need to consider when choosing a data type, and key aspects of

working with functions, usually in the context of certain types of data, like date and time data or

character data. For details and technicalities about data types, see Books Online for SQL Server

2012, under the topic “Data Types (Transact-SQL)” at *http://msdn.microsoft.com/en-us/library*

*/ms187752(v=SQL.110).aspx*. For details about built-in functions, see the topic “Built-in Functions

(Transact-SQL)” at *http://msdn.microsoft.com/en-us/library/ms174318(v=SQL.110).aspx*.

After this lesson, you will be able to:

■■ Choose the appropriate data type.

■■ Choose a type for your keys.

■■ Work with date and time, in addition to character data.

■■ Work with the CASE expression and related functions.

Estimated lesson time: 50 minutes

Choosing the Appropriate Data Type

Choosing the appropriate data types for your attributes is probably one of the most important

decisions that you will make regarding your data. SQL Server supports many data types

from different categories: exact numeric (INT, NUMERIC), character strings (CHAR, VARCHAR),

Unicode character strings (NCHAR, NVARCHAR), approximate numeric (FLOAT, REAL), binary

strings (BINARY, VARBINARY), date and time (DATE, TIME, DATETIME2, SMALLDATETIME,

DATETIME,

DATETIMEOFFSET), and others. There are many options, so it might seem like

a difficult task, but as long as you follow certain principles, you can be smart about your

choices, which results in a robust, consistent, and efficient database.

One of the great strengths of the relational model is the importance it gives to enforcement

of data integrity as part of the model itself, at multiple levels. One important aspect in

choosing the appropriate type for your data is to remember that a type is a constraint. This

means that it has a certain domain of supported values and will not allow values outside that

domain. For example, the DATE type allows only valid dates. An attempt to enter something

that isn’t a date, like 'abc' or '20120230', is rejected. If you have an attribute that is supposed

to represent a date, such as birthdate, and you use a type such as INT or CHAR, you don’t

benefit from built-in validating of dates. An INT type won’t prevent a value such as 99999999

and a CHAR type won’t prevent a value such as '20120230'.

Much like a type is a constraint, NOT NULL is a constraint as well. If an attribute isn’t supposed

to allow NULLs, it’s important to enforce a NOT NULL constraint as part of its definition.

Otherwise, NULLs will find their way into your attribute.

Also, you want to make sure that you do not confuse the formatting of a value with its

type. Sometimes, people use character strings to store dates because they think of storing a

date in a certain format. The formatting of a value is supposed to be the responsibility of the

application when data is presented. The type is a property of the value stored in the database,

and the internal storage format shouldn’t be your concern. This aspect has to do with an

important principle in the relational model called *physical data independence*.

A data type encapsulates behavior. By using an inappropriate type, you miss all the behavior

that is encapsulated in the type in the form of operators and functions that support it. As

a simple example, for types representing numbers, the plus (+) operator represents addition,

but for character strings, the same operator represents concatenation. If you chose an inappropriate

type for your value, you sometimes have to convert the type (explicitly or implicitly),

and sometimes juggle the value quite a bit, in order to treat it as what it is supposed to be.

Another important principle in choosing the appropriate type for your data is size. Often

one of the major aspects affecting query performance is the amount of I/O involved. A query

that reads less simply tends to run faster. The bigger the type that you use, the more storage

it uses. Tables with many millions of rows, if not billions, are commonplace nowadays.

When you start multiplying the size of a type by the number of rows in the table, the numbers

can quickly become significant. As an example, suppose you have an attribute representing

test scores, which are integers in the range 0 to 100. Using an INT data type for this

purpose is overkill. It would use 4 bytes per value, whereas a TINYINT would use only 1 byte,

and is therefore the more appropriate type in this case. Similarly, for data that is supposed

to represent dates, people have a tendency to use DATETIME, which uses 8 bytes of storage.

If the value is supposed to represent a date without a time, you should use DATE, which

uses only 3 bytes of storage. If the value is supposed to represent both date and time, you

should consider DATETIME2 or SMALLDATETIME. The former requires storage between 6 to 8

bytes (depending on precision), and as an added value, provides a wider range of dates and

improved, controllable precision. The latter uses only 4 bytes of storage, so as long as its supported

range of dates and precision cover your needs, you should use it. In short, you should

use the smallest type that serves your needs. Though of course, this applies not in the short

run, but in the long run. For example, using an INT type for a key in a table that at one point

or another will grow to a degree of billions of rows is a bad idea. You should be using BIGINT.

But using INT for an attribute representing test scores or DATETIME for date and time values

that require a minute precision are both bad choices even when thinking about the long run.

Be very careful with the imprecise types FLOAT and REAL. The first two sentences in the

documentation describing these types should give you a good sense of their nature: “Approximate-

number data types for use with floating point numeric data. Floating point data

is approximate; therefore, not all values in the data type range can be represented exactly.”

(You can find this documentation in the Books Online for SQL Server 2012 article “float and

real [Transact-SQL]” at *http://msdn.microsoft.com/en-us/library/ms173773.aspx*.) The benefit

in these types is that they can represent very large and very small numbers beyond what any

other numeric type that SQL Server supports can represent. So, for example, if you need to

represent very large or very small numbers for scientific purposes and don’t need complete

accuracy, you may find these types useful. They’re also quite economic (4 bytes for REAL and

8 bytes for FLOAT). But do not use them for things that are supposed to be precise.

*Real World* **Float Trouble**

We remember a case where a customer used FLOAT to represent barcode numbers of

products, and was then surprised by not getting the right product when scanning the

products’ barcodes. Also, recently, we got a query about conversion of a FLOAT value to

NUMERIC, resulting in a different value than what was entered. Here’s the case.

DECLARE @f AS FLOAT = '29545428.022495';

SELECT CAST(@f AS NUMERIC(28, 14)) AS value;

Can you guess what the output of this code is? Here it is.

Value

---------------------------------------

29545428.02249500200000

As mentioned, some values cannot be represented precisely.

In short, make sure you use exact numeric types when you need to represent values

precisely, and reserve the use of the approximate numeric types only to cases where you’re

certain that it’s acceptable for the application.

Another important aspect in choosing a type has to do with choosing fixed types (CHAR,

NCHAR, BINARY) vs. dynamic ones (VARCHAR, NVARCHAR, VARBINARY). Fixed types use the

storage for the indicated size; for example, CHAR(30) uses storage for 30 characters, whether

you actually specify 30 characters or less. This means that updates will not require the row to

physically expand, and therefore no data shifting is required. So for attributes that get updated

frequently, where the update performance is a priority, you should consider fixed types. Note

that when compression is used—specifically row compression—SQL Server stores fixed types

like variable ones, but with less overhead.

Variable types use the storage for what you enter, plus a couple of bytes for offset information

(or 4 bits with row compression). So for widely varying sizes of strings, if you use variable

types you can save a lot of storage. As already mentioned, the less storage used, the less

there is for a query to read, and the faster the query can perform. So variable length types

are usually preferable in such cases when read performance is a priority.

With character strings, there’s also the question of using regular character types (CHAR,

VARCHAR) vs. Unicode types (NCHAR, NVARCHAR). The former use 1 byte of storage per

character and support only one language (based on collation properties) besides English. The

latter use 2 bytes of storage per character (unless compressed) and support multiple languages.

If a surrogate pair is needed, a character will require 4 bytes of storage. So if data is

in multiple languages and you need to represent only one language besides English in your

data, you can benefit from using regular character types, with lower storage requirements.

When data is international, or your application natively works with Unicode data, you should

use Unicode data types so you don’t lose information. The greater storage requirements of

Unicode data are mitigated starting with SQL Server 2008 R2 with Unicode compression.

When using types that can have a length associated with them, such as CHAR and VARCHAR,

T-SQL supports omitting the length and then uses a default length. However, in different contexts,

the defaults can be different. It is considered a best practice to always be explicit about

the length, as in CHAR(1) or VARCHAR(30).

When defining attributes that represent the same thing across different tables—especially

ones that will later be used as join columns (like the primary key in one table and the foreign

key in another)—it’s very important to be consistent with the types. Otherwise, when comparing

one attribute with another, SQL Server has to apply implicit conversion of one attribute’s

type to the other, and this could have negative performance implications, like preventing efficient

use of indexes.

You also want to make sure that when indicating a literal of a type, you use the correct form.

For example, literals of regular character strings are delimited with single quotation marks, as in

'abc', whereas literals of Unicode character strings are delimited with a capital N and then single

quotation marks, as in N'abc'. When an expression involves elements with different types, SQL

Server needs to apply implicit conversion when possible, and this may result in performance

penalties. Note that in some cases the interpretation of a literal may not be what you think intuitively.

In order to force a literal to be of a certain type, you may need to apply explicit conversion

with functions like CAST, CONVERT, PARSE, or TRY\_CAST, TRY\_CONVERT, and TRY\_PARSE.

As an example, the literal 1 is considered an INT by SQL Server in any context. If you need the

literal 1 to be considered, for example, a BIT, you need to convert the literal’s type explicitly, as

in CAST(1 AS BIT). Similarly, the literal 4000000000 is considered NUMERIC and not BIGINT. If

you need the literal to be the latter, use CAST(4000000000 AS BIGINT). The difference between

the functions without the TRY and their counterparts with the TRY is that those without the TRY

fail if the value isn’t convertible, whereas those with the TRY return a NULL in such a case. For

example, the following code fails.

SELECT CAST('abc' AS INT);

Conversely, the following code returns a NULL.

SELECT TRY\_CAST('abc' AS INT);

As for the difference between CAST, CONVERT, and PARSE, with CAST, you indicate the

expression and the target type; with CONVERT, there’s a third argument representing the

style for the conversion, which is supported for some conversions, like between character

strings and date and time values. For example, CONVERT(DATE, '1/2/2012', 101) converts the

literal character string to DATE using style 101 representing the United States standard. With

PARSE, you can indicate the culture by using any culture supported by the Microsoft .NET

Framework. For example, PARSE('1/2/2012' AS DATE USING 'en-US') parses the input literal as

a DATE by using a United States English culture.

When using expressions that involve operands of different types, SQL Server usually converts

the one that has the lower data type precedence to the one with the higher. Consider

the expression 1 + '1' as an example. One operand is INT and the other is VARCHAR. If you

look in Books Online for SQL Server 2012, under “Data Type Precedence (Transact-SQL),” at

*http://msdn.microsoft.com/en-us/library/ms190309.aspx*, you will find that INT precedes VARCHAR;

hence, SQL Server implicitly converts the VARCHAR value '1' to the INT value 1, and the

result of the expression is therefore 2 and not the string '11'. Of course, you can always take

control by using explicit conversion.

If all operands of the expression are of the same type, that’s also going to be the type of

the result, and you might not want it to be the case. For example, the result of the expression

5 / 2 in T-SQL is the INT value 2 and not the NUMERIC value 2.5, because both operands are

integers, and therefore the result is an integer. If you were dealing with two integer columns,

like col1 / col2, and wanted the division to be NUMERIC, you would need to convert the columns

explicitly, as in CAST(col1 AS NUMERIC(12, 2)) / CAST(col2 AS NUMERIC(12, 2)).

Choosing a Data Type for Keys

When defining intelligent keys in your tables—namely keys based on already existing attributes

derived from the application—there’s no question about types because you already

chose those for your attributes. But when you need to create surrogate keys—ones that are

added solely for the purpose of being used as keys—you need to determine an appropriate

type for the attribute in addition to a mechanism to generate the key values. The reality

is that you will hear many different opinions as to what is the best solution—some based on

theory, and some backed by empirical evidence. But different systems and different workloads

could end up with different optimal solutions. What’s more, in some systems, write

performance might be the priority, whereas in others, the read performance is. One solution

can make the inserts faster but the reads slower, and another solution might work the other

way around. At the end of the day, to make smart choices, it’s important to learn the theory,

learn about others’ experiences, but eventually make sure that you run benchmarks in the

target system. In this respect, a sentence in a book called *Bubishi* by Patrick McCarthy (Tuttle

Publishing, 2008) is very fitting. It says, “Wisdom is putting knowledge into action.”

Note that this section refers to elements like sequence objects, the identity column property,

and indexes, which are covered in more detail later in the book. Chapter 11, “Other Data

Modification Aspects,” covers sequence objects and the identity property, and Chapter 15,

“Implementing Indexes and Statistics,” covers indexes. You may want to revisit this section

after finishing those chapters.

The typical options people use to generate surrogate keys are:

■■ **The identity column property** A property that automatically generates keys in

an attribute of a numeric type with a scale of 0; namely, any integer type (TINYINT,

SMALLINT, INT, BIGINT) or NUMERIC/DECIMAL with a scale of 0.

■■ **The sequence object** An independent object in the database from which you can

obtain new sequence values. Like identity, it supports any numeric type with a scale

of 0. Unlike identity, it’s not tied to a particular column; instead, as mentioned, it is an

independent object in the database. You can also request a new value from a sequence

object before using it. There are a number of other advantages over identity that will

be covered in Chapter 11.

■■ **Nonsequential GUI Ds** You can generate nonsequential global unique identifiers to

be stored in an attribute of a UNIQUEIDENTIFIER type. You can use the T-SQL function

NEWID to generate a new GUID, possibly invoking it with a default expression attached

to the column. You can also generate one from anywhere—for example, the client—

by using an application programming interface (API) that generates a new GUID. The

GUIDs are guaranteed to be unique across space and time.

■■ **Sequential GUI Ds** You can generate sequential GUIDs within the machine by using

the T-SQL function NEWSEQUENTIALID.

■■ **Custom solutions** If you do not want to use the built-in tools that SQL Server provides

to generate keys, you need to develop your own custom solution. The data type

for the key then depends on your solution.

*Exam Tip*

Understanding the built-in tools T-SQL provides for generating surrogate keys like the

sequence object, identity column property, and the NEWID and NEWSEQUENTIALID

functions, and their impact on performance, is an important skill for the exam.

One thing to consider regarding your choice of surrogate key generator and the data

type involved is the size of the data type. The bigger the type, the more storage is required,

and hence the slower the reads are. A solution using an INT data type requires 4 bytes per

value, BIGINT requires 8 bytes, UNIQUEIDENTIFIER requires 16 bytes, and so on. The storage

requirements for your surrogate key can have a cascading effect if your clustered index

is defined on the same key columns (the default for a primary key constraint). The clustered

index key columns are used by all nonclustered indexes internally as the means to locate rows

in the table. So if you define a clustered index on a column x, and nonclustered indexes—one

on column a, one on b, and one on c—your nonclustered indexes are internally created on

column (a, x), (b, x), and (c, x), respectively. In other words, the effect is multiplied.

Regarding the use of sequential keys (as with identity, sequence, and NEWSEQUENTIALID)

vs. nonsequential ones (as with NEWID or a custom randomized key generator), there are

several aspects to consider.

Starting with sequential keys, all rows go into the right end of the index. When a page is

full, SQL Server allocates a new page and fills it. This results in less fragmentation in the index,

which is beneficial for read performance. Also, insertions can be faster when a single session

is loading the data, and the data resides on a single drive or a small number of drives.

However, with high-end storage subsystems that have many spindles, the situation can be

different. When loading data from multiple sessions, you will end up with page latch contention

(*latches* are objects used to synchronize access to database pages) against the rightmost

pages of the index leaf level’s linked list. This bottleneck prevents use of the full throughput

of the storage subsystem.

Note that if you decide to use sequential keys and you’re using numeric ones, you can

always start with the lowest value in the type to use the entire range. For example, instead of

starting with 1 in an INT type, you could start with -2,147,483,648.

Consider nonsequential keys, such as random ones generated with NEWID or with a

custom solution. When trying to force a row into an already full page, SQL Server performs a

classic page split—it allocates a new page and moves half the rows from the original page to

the new one. A page split has a cost, plus it results in index fragmentation. Index fragmentation

can have a negative impact on the performance of reads. However, in terms of insert

performance, if the storage subsystem contains many spindles and you’re loading data from

multiple sessions, the random order can actually be better than sequential despite the splits.

That’s because there’s no hot spot at the right end of the index, and you use the storage subsystem’s

available throughput better. A good example for a benchmark demonstrating this

strategy can be found in a blog by Thomas Kejser at *http://blog.kejser.org/2011/10/05*

*/boosting-insert-speed-by-generating-scalable-keys/*.

Note that splits and index fragmentation can be mitigated by periodic index rebuilds as

part of the usual maintenance activities—assuming you have a window available for this.

If for aforementioned reasons you decide to rely on keys generated in random order, you

will still need to decide between GUIDs and a custom random key generator solution. As already

mentioned, GUIDs are stored in a UNIQUEIDENTIFIER type that is 16 bytes in size; that’s

large. But one of the main benefits of GUIDs is the fact that they can be generated anywhere

and not conflict across time and space. You can generate GUIDs not just in SQL Server using

the NEWID function, but anywhere, using APIs. Otherwise, you could come up with a custom

solution that generates smaller random keys. The solution can even be a mix of a built-in tool

and some tweaking on top. For example, you can find a creative solution by Wolfgang 'Rick'

Kutschera at *http://dangerousdba.blogspot.com/*

*2011/10/day-sequences-saved-world.html*.

Rick uses the SQL Server sequence object, but flips

the bits of the values so that the insertion

is distributed across the index leaf.

To conclude this section about keys and types for keys, remember that there are multiple

options. Smaller is generally better, but then there’s the question of the hardware that you

use, and where your performance priorities are. Also remember that although it is very important

to make educated guesses, it is also important to benchmark solutions in the target

environment.

Date and Time Functions

T-SQL supports a number of date and time functions that allow you to manipulate your date

and time data. Support for date and time functions keeps improving, with the last two versions

of SQL Server adding a number of new functions.

This section covers some of the important functions supported by T-SQL and provides

some examples. For the full list, as well as the technical details and syntax elements, see

Books Online for SQL Server 2012, under the topic “Date and Time Data Types and Functions

(Transact-SQL)” at *http://msdn.microsoft.com/en-us/library/ms186724(v=SQL.110).aspx*.

Current Date and Time

One important category of functions is the category that returns the current date and time.

The functions in this category are GETDATE, CURRENT\_TIMESTAMP, GETUTCDATE, SYSDATETIME,

SYSUTCDATETIME, and SYSDATETIMEOFFSET.

GETDATE is T-SQL–specific, returning the current date and time in the SQL Server instance

you’re connected to as a DATETIME data type. CURRENT\_TIMESTAMP is the same, only it’s

standard, and hence the recommended one to use. SYSDATETIME and SYSDATETIMEOFFSET

are similar, only returning the values as the more precise DATETIME2 and DATETIMEOFFSET

types (including offset), respectively. Note that there are no built-in functions to return

the current date or the current time; to get such information, simply cast the SYSDATETIME

function to DATE or TIME, respectively. For example, to get the current date, use

CAST(SYSDATETIME() AS DATE). The GETUTCDATE function returns the current date and time

in UTC terms as a DATETIME type, and SYSUTCDATE does the same, only returning the result

as the more precise DATETIME2 type.

Date and Time Parts

This section covers date and time functions that either extract a part from a date and time

value (like DATEPART) or construct a date and time value from parts (like DATEFROMPARTS).

Using the DATEPART function, you can extract from an input date and time value a desired

part, such as a year, minute, or nanosecond, and return the extracted part as an integer. For

example, the expression DATEPART(month, '20120212') returns 2. T-SQL provides the functions

YEAR, MONTH, and DAY as abbreviations to DATEPART, not requiring you to specify the

part. The DATENAME function is similar to DATEPART, only it returns the name of the part

as a character string, as opposed to the integer value. Note that the function is languagedependent.

That is, if the effective language in your session is us\_english, the expression

DATENAME(month, '20120212') returns 'February', but for Italian, it returns 'febbraio'.

T-SQL provides a set of functions that construct a desired date and time value from its numeric

parts. You have such a function for each of the six available date and time types: DATEFROMPARTS,

DATETIME2FROMPARTS, DATETIMEFROMPARTS, DATETIMEOFFSETFROMPARTS,

SMALLDATETIMEFROMPARTS, and TIMEFROMPARTS. For example, to build a DATE value from

its parts, you would use an expression such as DATEFROMPARTS(2012, 02, 12).

Finally, the EOMONTH function computes the respective end of month date for the input

date and time value. For example, suppose that today was February 12, 2012. The expression

EOMONTH(SYSDATETIME()) would then return the date '2012-02-29'. This function supports a

second optional input indicating how many months to add to the result.

Add and Diff

T-SQL supports addition and difference date and time functions called DATEADD and DATEDIFF.

DATEADD is a very commonly used function. With it, you can add a requested number

of units of a specified part to a specified date and time value. For example, the expression

DATEADD(year, 1, '20120212') adds one year to the input date February 12, 2012.

DATEDIFF is another commonly used function; it returns the difference in terms of a requested

part between two date and time values. For example, the expression DATEDIFF(day,

'20110212', '20120212') computes the difference in days between February 12, 2011 and

February 12, 2012, returning the value 365. Note that this function looks only at the parts

from the requested one and above in the date and time hierarchy—not below. For example,

the expression DATEDIFF(year, '20111231', '20120101') looks only at the year part, and hence

returns 1. It doesn’t look at the month and day parts of the values.

Offset

T-SQL supports two functions related to date and time values with an offset: SWITCHOFFSET

and TODATETIMEOFFSET.

With the SWITCHOFFSET function, you can return an input DATETIMEOFFSET value in a

requested offset term. For example, consider the expression SWITCHOFFSET(SYSDATETIMEOF

FSET(), '-08:00'). Regardless of the offset of the instance you are connected to, you request to

present the current date and time value in terms of offset '-08:00'. If the system’s offset is, say,

'-05:00', the function will compensate for this by subtracting three hours from the input value.

The TODATETIMEOFFSET function is used for a different purpose. You use it to construct a

DATETIMEOFFSET value from two inputs: the first is a date and time value that is not offsetaware,

and the second is the offset. You can use this function when migrating from data that

is not offset-aware, where you keep the local date and time value in one attribute, and the

offset in another, to offset-aware data. Say you have the local date and time in an atribute

called dt, and the offset in an attribute called theoffset. You add an attribute called dto of

a DATETIMEOFFSET type to the table. You then update the new attribute to the expression

TODATETIMEOFFSET(dt, theoffset), and then drop the original attributes dt and theoffset

from the table.

The following code demonstrates using both functions.

SELECT

SWITCHOFFSET('20130212 14:00:00.0000000 -08:00', '-05:00') AS [SWITCHOFFSET],

TODATETIMEOFFSET('20130212 14:00:00.0000000', '-08:00') AS [TODATETIMEOFFSET];

Here’s the output of this code.

SWITCHOFFSET TODATETIMEOFFSET

---------------------------------- ----------------------------------

2013-02-12 17:00:00.0000000 -05:00 2013-02-12 14:00:00.0000000 -08:00

Character Functions

T-SQL was not really designed to support very sophisticated character string manipulation

functions, so you won’t find a very large set of such functions. This section describes the character

string functions that T-SQL does support, arranged in categories.

Concatenation

Character string concatenation is a very common need. T-SQL supports two ways to concatenate

strings—one with the plus (+) operator, and another with the CONCAT function.

Here’s an example for concatenating strings in a query by using the + operator.

SELECT empid, country, region, city,

country + N',' + region + N',' + city AS location

FROM HR.Employees;

Here’s the result of this query.

empid country region city location

------ -------- ------- --------- ----------------

1 USA WA Seattle USA,WA,Seattle

2 USA WA Tacoma USA,WA,Tacoma

3 USA WA Kirkland USA,WA,Kirkland

4 USA WA Redmond USA,WA,Redmond

5 UK NULL London NULL

6 UK NULL London NULL

7 UK NULL London NULL

8 USA WA Seattle USA,WA,Seattle

9 UK NULL London NULL

Observe that when any of the inputs is NULL, the + operator returns a NULL. That’s

standard behavior that can be changed by turning off a session option called CONCAT\_NULL\_

YIELDS\_NULL\_INPUT, though it’s not recommended to rely on nonstandard behavior. If you

want to substitute a NULL with an empty string, there are a number of ways for you to do this

programmatically. One option is to use COALESCE(<*expression*>, ''). For example, in this data,

only region can be NULL, so you can use the following query to replace a comma plus region

with an empty string when region is NULL.

SELECT empid, country, region, city,

country + COALESCE( N',' + region, N'') + N',' + city AS location

FROM HR.Employees;

Another option is to use the CONCAT function which, unlike the + operator, substitutes a

NULL input with an empty string. Here’s how the query looks.

SELECT empid, country, region, city,

CONCAT(country, N',' + region, N',' + city) AS location

FROM HR.Employees;

Here’s the output of this query.

empid country region city location

------ -------- ------- --------- ----------------

1 USA WA Seattle USA,WA,Seattle

2 USA WA Tacoma USA,WA,Tacoma

3 USA WA Kirkland USA,WA,Kirkland

4 USA WA Redmond USA,WA,Redmond

5 UK NULL London UK,London

6 UK NULL London UK,London

7 UK NULL London UK,London

8 USA WA Seattle USA,WA,Seattle

9 UK NULL London UK,London

Observe that this time, when region was NULL, it was replaced with an empty string.

Substring Extraction and Position

This section covers functions that you can use to extract a substring from a string, and identify

the position of a substring within a string.

With the SUBSTRING function, you can extract a substring from a string given as the first

argument, starting with the position given as the second argument, and a length given as the

third argument. For example, the expression SUBSTRING('abcde', 1, 3) returns 'abc'. If the third

argument is greater than what would get you to the end of the string, the function doesn’t

fail; instead, it just extracts the substring until the end of the string.

The LEFT and RIGHT functions extract a requested number of characters from the left and

right ends of the input string, respectively. For example, LEFT('abcde', 3) returns 'abc' and

RIGHT('abcde', 3) returns 'cde'.

The CHARINDEX function returns the position of the first occurrence of the string provided

as the first argument within the string provided as the second argument. For example, the expression

CHARINDEX(' ','Itzik Ben-Gan') looks for the first occurrence of a space in the second

input, returning 6 in this example. Note that you can provide a third argument indicating to

the function where to start looking.

You can combine, or nest, functions in the same expression. For example, suppose you

query a table with an attribute called fullname formatted as '<*first*> <*last*>', and you need to

write an expression that extracts the first name part. You can use the following expression.

LEFT(fullname, CHARINDEX(' ', fullname) - 1)

T-SQL also supports a function called PATINDEX that, like CHARINDEX, you can use to

locate the first position of a string within another string. But whereas with CHARINDEX you’re

looking for a constant string, with PATINDEX you’re looking for a pattern. The pattern is

formed very similar to the LIKE patterns that you’re probably familiar with, where you use

wildcards like % for any string, \_ for a single character, and square brackets ([]) representing a

single character from a certain list or range. If you’re not familiar with such pattern construction,

see the topics “PATINDEX (Transact-SQL)” and “LIKE (Transact-SQL)” in Books Online

for SQL Server 2012 at *http://msdn.microsoft.com/en-us/library/ms188395(v=SQL.110).aspx*

and *http://msdn.microsoft.com/en-us/library/ms179859(v=SQL.110).aspx*. As an example, the

expression PATINDEX('%[0-9]%', 'abcd123efgh') looks for the first occurrence of a digit (a

character in the range 0–9) in the second input, returning the position 6 in this case.

String Length

T-SQL provides two functions that you can use to measure the length of an input value—

LEN and DATALENGTH.

The LEN function returns the length of an input string in terms of the number of characters.

Note that it returns the number of characters, not bytes, whether the input is a regular

character or Unicode character string. For example, the expression LEN(N'xyz') returns 3. If

there are any trailing spaces, LEN removes them.

The DATALENGTH function returns the length of the input in terms of number of bytes.

This means, for example, that if the input is a Unicode character string, it will count 2 bytes

per character. For example, the expression DATALENGTH(N'xyz') returns 6. Note also that,

unlike LEN, the DATALENGTH function doesn’t remove trailing spaces.

String Alteration

T-SQL supports a number of functions that you can use to apply alterations to an input string.

Those are REPLACE, REPLICATE, and STUFF.

With the REPLACE function, you can replace in an input string provided as the first argument

all occurrences of the string provided as the second argument, with the string provided

as the third argument. For example, the expression REPLACE('.1.2.3.', '.', '/') substitutes all occurrences

of a dot (.) with a slash (/), returning the string '/1/2/3/'.

The REPLICATE function allows you to replicate an input string a requested number of

times. For example, the expression REPLICATE('0', 10) replicates the string '0' ten times, returning

'0000000000'.

The STUFF function operates on an input string provided as the first argument; then, from

the character position indicated as the second argument, deletes the number of characters

indicated by the third argument. Then it inserts in that position the string specified as the

fourth argument. For example, the expression STUFF(',x,y,z', 1, 1, '') removes the first character

from the input string, returning 'x,y,z'.

String Formatting

This section covers functions that you can use to apply formatting options to an input string.

Those are the UPPER, LOWER, LTRIM, RTRIM, and FORMAT functions.

The first four functions are self-explanatory (uppercase form of the input, lowercase form

of the input, input after removal of leading spaces, and input after removal of trailing spaces).

Note that there’s no TRIM function that removes both leading and trailing spaces; to achieve

this, you need to nest one function call within another, as in RTRIM(LTRIM(<input>)).

With the FORMAT function, you can format an input value based on a format string, and

optionally specify the culture as a third input where relevant. You can use any format string

supported by the .NET Framework. (For details, see the topics “FORMAT (Transact-SQL)” and

“Formating Types” at *http://msdn.microsoft.com/en-us/library/hh213505(v=sql.110).aspx* and

*http://msdn.microsoft.com/en-us/library/26etazsy.aspx*.) As an example, the expression FORMAT(

1759, '000000000') formats the input number as a character string with a fixed size of 10

characters with leading zeros, returning '0000001759'.

CASE Expression and Related Functions

T-SQL supports an expression called CASE and a number of related functions that you can use

to apply conditional logic to determine the returned value. Many people incorrectly refer to

CASE as a statement. A statement performs some kind of an action or controls the flow of the

code, and that’s not what CASE does; CASE returns a value, and hence is an expression.

The CASE expression has two forms—the *simple* form and the *searched* form. Here’s an

example of the simple CASE form issued against the sample database TSQL2012.

SELECT productid, productname, unitprice, discontinued,

CASE discontinued

WHEN 0 THEN 'No'

WHEN 1 THEN 'Yes'

ELSE 'Unknown'

END AS discontinued\_desc

FROM Production.Products;

The simple form compares an *input expression* (in this case the attribute discontinued)

to multiple possible scalar *when expressions* (in this case, 0 and 1), and returns the *result*

*expression* (in this case, 'No' and 'Yes', respectively) associated with the first match. If there’s

no match and an ELSE clause is specified, the *else expression* (in this case, 'Unknown') is returned.

If there’s no ELSE clause, the default is ELSE NULL. Here’s an abbreviated form of the

output of this query.

productid productname unitprice discontinued discontinued\_desc

---------- -------------- ---------- ------------ -----------------

1 Product HHYDP 18.00 0 No

2 Product RECZE 19.00 0 No

3 Product IMEHJ 10.00 0 No

4 Product KSBRM 22.00 0 No

5 Product EPEIM 21.35 1 Yes

...

The searched form of the CASE expression is more flexible. Instead of comparing an input

expression to multiple possible expressions, it uses predicates in the WHEN clauses, and the

first predicate that evaluates to true determines which when expression is returned. If none is

true, the CASE expression returns the else expression. Here’s an example.

SELECT productid, productname, unitprice,

CASE

WHEN unitprice < 20.00 THEN 'Low'

WHEN unitprice < 40.00 THEN 'Medium'

WHEN unitprice >= 40.00 THEN 'High'

ELSE 'Unknown'

END AS pricerange

FROM Production.Products;

In this example, the CASE expression returns a description of the product’s unit price

range. When the unit price is below $20.00, it returns 'Low', when it’s $20.00 or more and

below $40.00, it returns 'Medium', and when it’s $40.00 or more, it returns 'High'. There’s an

ELSE clause for safety; if the input is NULL, the else expression returned is 'Unknown'. Notice

that the second when predicate didn’t need to check whether the value is $20.00 or more

explicitly. That’s because the when predicates are evaluated in order and the first when predicate

did not evaluate to true. Here’s an abbreviated form of the output of this query.

productid productname unitprice pricerange

---------- -------------- ---------- ----------

1 Product HHYDP 18.00 Low

2 Product RECZE 19.00 Low

3 Product IMEHJ 10.00 Low

4 Product KSBRM 22.00 Medium

5 Product EPEIM 21.35 Medium

...

T-SQL supports a number of functions that can be considered as abbreviates of the CASE

expression. Those are the standard COALESCE and NULLIF functions, and the nonstandard

ISNULL, IIF, and CHOOSE

The COALESCE function accepts a list of expressions as input and returns the first that

is not NULL, or NULL if all are NULLs. For example, the expression COALESCE(NULL, 'x', 'y')

returns 'x'. More generally, the expression:

COALESCE(<exp1>, <exp2>, …, <expn>)

is similar to the following.

CASE

WHEN <exp1> IS NOT NULL THEN <exp1>

WHEN <exp2> IS NOT NULL THEN <exp2>

…

WHEN <expn> IS NOT NULL THEN <expn>

ELSE NULL

END

A typical use of COALESCE is to substitute a NULL with something else. For example, the

expression COALESCE(region, '') returns region if it’s not NULL and returns an empty string if

it is NULL.

T-SQL supports a nonstandard function called ISNULL that is similar to the standard

COALESCE, but it’s a bit more limited in the sense that it supports only two inputs. Like

COALESCE, it returns the first input that is not NULL. So, instead of COALESCE(region, ''), you

could use ISNULL(region, ''). Generally, it is recommended to stick to standard features unless

there’s some flexibility or performance advantage in the nonstandard feature that is a higher

priority. ISNULL is actually more limited than COALESCE, so generally, it is recommended to

stick to COALESCE.

There are a couple of subtle differences between COALESCE and ISNULL that you might be

interested in. One difference is in which input determines the type of the output. Consider the

following code.

DECLARE

@x AS VARCHAR(3) = NULL,

@y AS VARCHAR(10) = '1234567890';

SELECT COALESCE(@x, @y) AS [COALESCE], ISNULL(@x, @y) AS [ISNULL];

Here’s the output of this code.

COALESCE ISNULL

---------- ------

1234567890 123

Observe that the type of the COALESCE expression is determined by the returned element,

whereas the type of the ISNULL expression is determined by the first input.

The other difference between COALESCE and ISNULL is when you are using SELECT INTO,

which is discussed in more detail in Chapter 11. Suppose the SELECT list of a SELECT INTO

statement contains the expressions COALESCE(col1, 0) AS newcol1 vs. ISNULL(col1, 0) AS

newcol1. If the source attribute col1 is defined as NOT NULL, both expressions will produce

an attribute in the result table defined as NOT NULL. However, if the source attribute col1 is

defined as allowing NULLs, COALESCE will create a result attribute allowing NULLs, whereas

ISNULL will create one that disallows NULLs.

*Exam Tip*

COALESCE and ISNULL can impact performance when you are combining sets; for example,

with joins or when you are filtering data. Consider an example where you have two tables

T1 and T2 and you need to join them based on a match between T1.col1 and T2.col1. The

attributes do allow NULLs. Normally, a comparison between two NULLs yields unknown,

and this causes the row to be discarded. You want to treat two NULLs as equal. What

some do in such a case is use COALESCE or ISNULL to substitute a NULL with a value that

they know cannot appear in the data. For example, if the attributes are integers, and you

know that you have only positive integers in your data (you can even have constraints that

ensure this), you might try to use the predicate COALESCE(T1.col1, -1) = COALESCE(T2.

col1, -1), or ISNULL(T1.col1, -1) = ISNULL(T2.col1, -1). The problem with this form is that,

because you apply manipulation to the attributes you’re comparing, SQL Server will not

rely on index ordering. This can result in not using available indexes efficiently. Instead, it

is recommended to use the longer form: T1.col1 = T2.col1 OR (T1.col1 IS NULL AND T2.col1

IS NULL), which SQL Server understands as just a comparison that considers NULLs as

equal. With this form, SQL Server can efficiently use indexing.

T-SQL also supports the standard NULLIF function. This function accepts two input expressions,

returns NULL if they are equal, and returns the first input if they are not. For example,

consider the expression NULLIF(col1, col2). If col1 is equal to col2, the function returns a

NULL; otherwise, it returns the col1 value.

As for IIF and CHOOSE, these are nonstandard T-SQL functions that were added to simplify

migrations from Microsoft Access platforms. Because these functions aren’t standard and

there are simple standard alternatives with CASE expressions, it is not usually recommended

that you use them. However, when you are migrating from Access to SQL Server, these functions

can help with smoother migration, and then gradually you can refactor your code to use

the available standard functions. With the IIF function, you can return one value if an input

predicate is true and another value otherwise. The function has the following form.

IIF(<predicate>, <true\_result>, <false\_or\_unknown\_result>)

This expression is equivalent to the following.

CASE WHEN <predicate> THEN <true\_result> ELSE <false\_or\_unknown\_result> END

For example, the expression IIF(orderyear = 2012, qty, 0) returns the value in the qty attribute

when the orderyear attribute is equal to 2012, and zero otherwise.

The CHOOSE function allows you to provide a position and a list of expressions, and returns

the expression in the indicated position. The function takes the following form.

CHOOSE(<pos>, <exp1>, <exp2>, …, <expn>)

For example, the expression CHOOSE(2, 'x', 'y', 'z') returns 'y'. Again, it’s straightforward

to replace a CHOOSE expression with a logically equivalent CASE expression; but the point

in supporting CHOOSE, as well as IIF, is to simplify migrations from Access to SQL Server as a

temporary solution.

**Quick Check**

1. Would you use the type FLOAT to represent a product unit price?

2. What is the difference between NEWID and NEWSEQUENTIALID?

3. Which function returns the current date and time value as a DATETIME2 type?

4. When concatenating character strings, what is the difference between the plus

(+) operator and the CONCAT function?

**Quick Check Answer**

1. No, because FLOAT is an approximate data type and cannot represent all values

precisely.

2. The NEWID function generates GUID values in random order, whereas the

NEWSEQUENTIAL ID function generates GUIDs that increase in a sequential

order.

3. The SYSDATETIME function.

4. The + operator by default yields a NULL result on NULL input, whereas the

CONCAT function treats NULLs as empty strings.

Practice **Working with Data Types and Built-in Functions**

In this practice, you exercise your knowledge of data types and functions. You query data from

existing tables and manipulate existing attributes by using functions. You are provided with exercises

that contain requests to write queries that address certain tasks. It is recommended that

you first try to write the query yourself and then compare your answer with the given solution.

If you encounter a problem completing an exercise, you can install the completed projects

from the Solution folder that is provided with the companion content for this chapter and

lesson.

Exercise 1 A pply String Concatenation and Use a Date and Time Function

In this exercise, you practice string concatenation and the use of a date and time function.

1. Open SSMS and connect to the sample database TSQL2012.

2. Write a query against the HR.Employees table that returns the employee ID, the full

name of the employee (concatenate the attributes firstname, space, and lastname), and

the birth year (apply a function to the birthdate attribute). Here’s one possible query

that achieves this task.

SELECT empid,

firstname + N' ' + lastname AS fullname,

YEAR(birthdate) AS birthyear

FROM HR.Employees;

Exercise 2 Use Additional Date and Time Functions

In this exercise, you practice the use of additional date and time functions.

Write an expression that computes the date of the last day of the current month. Also write

an expression that computes the last day of the current year. Of course, there are a number of

ways to achieve such tasks. Here’s one way to compute the end of the current month.

SELECT EOMONTH(SYSDATETIME()) AS end\_of\_current\_month;

And here’s one way to compute the end of the current year.

SELECT DATEFROMPARTS(YEAR(SYSDATETIME()), 12, 31) AS end\_of\_current\_year;

Using the YEAR function, you extract the current year. Then provide the current year along

with the month 12 and the day 31 to the DATEFROMPARTS function to construct the last day

of the current year.

Exercise 3 Use String and Conversion Functions

In this exercise, you practice the use of string and conversion functions.

1. Write a query against the Production.Products table that returns the existing numeric

product ID, in addition to the product ID formatted as a fixed-sized string with 10

digits with leading zeros. For example, for product ID 42, you need to return the string

'0000000042'. One way to address this need is by using the following code.

SELECT productid,

RIGHT(REPLICATE('0', 10) + CAST(productid AS VARCHAR(10)), 10) AS str\_productid

FROM Production.Products;

2. Using the REPLICATE function, you generate a string made of 10 zeros. Next you

concatenate the character form of the product ID. Then you extract the 10 rightmost

characters from the result string.

Can you think of a simpler way to achieve the same task using new functions that were

introduced in SQL Server 2012? A much simpler way to achieve the same thing is by using

the FORMAT function, as in the following query.

SELECT productid,

FORMAT(productid, 'd10') AS str\_productid

FROM Production.Products;

Lesson Summary

■■ Your choices of data types for your attributes will have a dramatic effect on the functionality

and performance of the T-SQL code that interacts with the data—even more

so for attributes used as keys. Therefore, much care and consideration should be taken

when choosing types.

■■ T-SQL supports a number of functions that you can use to apply manipulation of date

and time data, character string data, and other types of data. Remember that T-SQL

was mainly designed to handle data manipulation, and not formatting and similar

needs. Therefore, in those areas, you will typically find only fairly basic support. Such

tasks are usually best handled in the client.

■■ T-SQL provides the CASE expression that allows you to return a value based on conditional

logic, in addition to a number of functions that can be considered abbreviations

of CASE.

Lesson Review

Answer the following questions to test your knowledge of the information in this lesson. You

can find the answers to these questions and explanations of why each answer choice is correct

or incorrect in the “Answers” section at the end of this chapter.

1. Why is it important to use the appropriate type for attributes?

A. Because the type of your attribute enables you to control the formatting of the

values

B. Because the type constrains the values to a certain domain of supported values

C. Because the type prevents duplicates

D. Because the type prevents NULLs

2. Which of the following functions would you consider using to generate surrogate keys?

(Choose all that apply.)

A. NEWID

B. NEWSEQUENTIALID

C. GETDATE

D. CURRENT\_TIMESTAMP

3. What is the difference between the simple CASE expression and the searched CASE

expression?

A. The simple CASE expression is used when the database recovery model is simple,

and the searched CASE expression is used when it’s full or bulk logged.

B. The simple CASE expression compares an input expression to multiple possible

expressions in the WHEN clauses, and the searched CASE expression uses independent

predicates in the WHEN clauses.

C. The simple CASE expression can be used anywhere in a query, and the searched

CASE expression can be used only in the WHERE clause.

D. The simple CASE expression can be used anywhere in a query, and the searched

CASE expression can be used only in query filters (ON, WHERE, HAVING).

**Case Scenarios**

In the following case scenarios, you apply what you’ve learned about the SELECT statement.

You can find the answers to these questions in the “Answers” section at the end of this chapter.

Case Scenario 1: Reviewing the Use of Types

You are hired as a consultant to help address performance issues in an existing system. The

system was developed originally by using SQL Server 2005 and has recently been upgraded

to SQL Server 2012. Write rates in the system are fairly low, and their performance is more

than adequate. Also, write performance is not a priority. However, read performance is a priority,

and currently it is not satisfactory. One of the main goals of the consulting engagement

is to provide recommendations that will help improve read performance. You have a meeting

with representatives of the customer, and they ask for your recommendations in different

potential areas for improvement. One of the areas they inquire about is the use of data types.

Your task is to respond to the following customer queries:

1. We have many attributes that represent a date, like order date, invoice date, and so on,

and currently we use the DATETIME data type for those. Do you recommend sticking

to the existing type or replacing it with another? Any other recommendations along

similar lines?

2. We have our own custom table partitioning solution because we’re using the Standard

edition of SQL Server. We use a surrogate key of a UNIQUEIDENTIFIER type with the

NEWID function invoked by a default constraint expression as the primary key for the

tables. We chose this approach because we do not want keys to conflict across the different

tables. This primary key is also our clustered index key. Do you have any recommendations

concerning our choice of a key?

Case Scenario 2: Reviewing the Use of Functions

The same company who hired you to review their use of data types would like you to also

review their use of functions. They pose the following question to you:

■■ Our application has worked with SQL Server so far, but due to a recent merger with

another company, we need to support other database platforms as well. What can you

recommend in terms of use of functions?

**Suggested Practices**

To help you successfully master the exam objectives presented in this chapter, complete the

following tasks.

Analyze the Data Types in the Sample Database

To practice your knowledge of data types, analyze the data types in the sample database

TSQL2012.

■■ **Practice 1** Using the Object Explorer in SSMS, navigate to the sample database

TSQL2012. Analyze the choices of the data types for the different attributes and try to

reason about the choices. Also, evaluate whether the choices made are optimal and

think about whether there’s any room for improvement in some cases.

■■ **Practice 2** Visit Books Online under “Data Type Precedence (Transact-SQL),” at *http://*

*msdn.microsoft.com/en-us/library/ms190309.aspx*. Identify the precedence order

among the types INT, DATETIME, and VARCHAR. Try to reason about Microsoft’s

choice of this precedence order.

Analyze Code Samples in Books Online for SQL Server 2012

To better understand the use of built-in functions, analyze and execute the code samples

provided in Books Online for SQL Server 2012.

■■ **Practice 1** Visit the Books Online article “Date and Time Data Types and Functions

(Transact-SQL)” at *http://msdn.microsoft.com/en-us/library/ms186724(v=SQL.110).aspx*.

From there, follow the links that lead to articles about individual functions that seem

useful to you. In those articles, go to the Examples section. Analyze those examples,

execute them, and make sure that you understand them.

■■ **Practice 2** Similar to Practice 1, go to the Books Online article “String Functions

(Transact-SQL)” at *http://msdn.microsoft.com/en-us/library/ms181984(v=SQL.110).aspx*.

Follow the links for functions that seem useful to you. In those articles, go to the Examples

section. Analyze and execute the examples, and make sure you understand them.

**Answers**

This section contains the answers to the lesson review questions and solutions to the case

scenarios in this chapter.

Lesson 1

1. **Correct Answers: B and D**

A. **Incorrect:** Attribute aliasing allows you to meet relational requirements, so it’s

certainly more than an aesthetic feature.

B. **Correct:** The relational model requires that all attributes have names.

C. **Incorrect:** T-SQL allows a result attribute to be without a name when the expression

is based on a computation without an alias.

D. **Correct:** You can assign your own name to a result attribute by using an alias.

2. **Correct Answer: C**

A. **Incorrect:** The FROM and SELECT clauses are mandatory in a SELECT query according

to standard SQL but not T-SQL.

B. **Incorrect:** The WHERE clause is optional in T-SQL.

C. **Correct:** According to T-SQL, the only mandatory clause is the SELECT clause.

D. **Incorrect:** The FROM and WHERE clauses are both optional in T-SQL.

3. **Correct Answers: C and D**

A. **Incorrect:** Aliasing columns with the AS clause is standard and considered a best

practice.

B. **Incorrect:** Aliasing tables with the AS clause is standard and considered a best

practice.

C. **Correct:** Not aliasing a column that is a result of a computation is nonrelational

and is considered a bad practice.

D. **Correct:** Using \* in the SELECT list is considered a bad practice.

Lesson 2

1. **Correct Answer: B**

A. **Incorrect:** Formatting isn’t a responsibility of the type or the data layer in general;

rather, it is the responsibility of the presentation layer.

B. **Correct:** The type should be considered a constraint because it limits the values

allowed.

C. **Incorrect:** The type itself doesn’t prevent duplicates. If you need to prevent duplicates,

you use a primary key or unique constraint.

D. **Incorrect:** A type doesn’t prevent NULLs. For this, you use a NOT NULL constraint.

2. **Correct Answers: A and B**

A. **Correct:** The NEWID function creates GUIDs in random order. You would consider

it when the size overhead is not a major issue and the ability to generate a unique

value across time and space, from anywhere, in random order is a higher priority.

B. **Correct:** The NEWSEQUENTIALID function generates GUIDs in increasing order

within the machine. It helps reduce fragmentation and works well when a single

session loads the data, and the number of drives is small. However, you should

carefully consider an alternative using another key generator, like a sequence object,

with a smaller type when possible.

C. **Incorrect:** There’s no assurance that GETDATE will generate unique values; therefore,

it’s not a good choice to generate keys.

D. **Incorrect:** The CURRENT\_TIMESTAMP function is simply the standard version of

GETDATE, so it also doesn’t guarantee uniqueness.

3. **Correct Answer: B**

A. **Incorrect:** CASE expressions have nothing to do with the database recovery

model.

B. **Correct:** The difference between the two is that the simple form compares

expressions and the searched form uses predicates.

C. **Incorrect:** Both CASE expressions are allowed wherever a scalar expression is

allowed—anywhere in the query.

D. **Incorrect:** Both CASE expressions are allowed wherever a scalar expression is

allowed—anywhere in the query.

Case Scenario 1

1. The DATETIME data type uses 8 bytes of storage. SQL Server 2012 supports the DATE

data type, which uses 3 bytes of storage. In all those attributes that represent a date

only, it is recommended to switch to using DATE. The lower the storage requirement,

the better the reads can perform.

As for other recommendations, the general rule “smaller is better, provided that you

cover the needs of the attribute in the long run” is suitable for read performance. For

example, if you have descriptions of varying lengths stored in a CHAR or NCHAR type,

consider switching to VARCHAR or NVARCHAR, respectively. Also, if you’re currently

using Unicode types but need to store strings of only one language—say, US English—

consider using regular characters instead.

2. For one, the UNIQUEIDENTIFIER type is large—16 bytes. And because it’s also the

clustered index key, it is copied to all nonclustered indexes. Also, due to the random

order in which the NEWID function generates values, there’s probably a high level

of fragmentation in the index. A different approach to consider (and test!) is switching

to an integer type and using the sequence object to generate keys that do not

conflict across tables. Due to the reduced size of the type, with the multiplied effect

on nonclustered indexes, performance of reads will likely improve. The values will be

increasing, and as a result, there will be less fragmentation, which will also likely have a

positive effect on reads.

Case Scenario 2

■■ To improve the portability of the code, it’s important to use standard code when possible,

and this of course applies more specifically to the use of built-in functions. For

example, use COALESCE and not ISNULL, use CURRENT\_TIMESTAMP and not GETDATE,

and use CASE and not IIF.