IN THIS CHAPTER

- Discovering valid values for table columns
- Summarizing data with set functions
- Dissecting data with value functions
- Converting data types

Chapter 1
Values, Variables, Functions, and Expressions

This chapter describes the tools that ISO/IEC standard SQL provides to operate on data. In addition to specifying the value of a data item, you can slice and dice an item in a variety of ways. Instead of just retrieving raw data as it exists in the database, you can preprocess it to deliver just the information you want, in the form that you want it.

Entering Data Values

After you've created a database table, the next step is to enter data into it. SQL supports a number of different data types. (Refer to Book 1, Chapter 6 for coverage of those types.) Within any specific data type, the data can take any of several forms. The five different forms that can appear in table rows are

- Row values
- Column references

- Literal values
- Variables
- Special variables

I discuss each in turn throughout this section.

Row values have multiple parts

A *row value* includes the values of all the data in all the columns in a row in a table. It is actually multiple values rather than just one. The intersection of a row and a column, called a *field,* contains a single, so-called "atomic" value. All the values of all the fields in a row, taken together, are that single row's row value.

Identifying values in a column

Just as you can specify a row value consisting of multiple values, you can specify the value contained in a single column. For illustration, consider this example from the Honest Abe database shown back in Book 2, Chapter 3:

```
SELECT * FROM CUSTOMER
   WHERE CustomerName = 'John Smith' ;
```

This query returns all the rows in the CUSTOMER table where the value in the CustomerName column is `John Smith`.

Literal values don't change

In SQL, a value can either be a constant or it can be represented by a variable. Constant values are called *literals*. Table 1-1 shows sample literals for each of the SQL data types.

REMEMBER

Numeric literals are just the values that they represent. Nonnumeric literals are enclosed in single quotes.

TIP

The SQL:2023 standard adds a couple of new features to use with numeric literal values. For large integer values, you can add underscores to make the value more readable (for example, `1_000_000`). You can also now specify nondecimal numeric literal values as binary, octal, or hexadecimal values. Binary values are preceded with `0b` (for example, `0b1011`), octal values with `0o` (for example, `0o672`), and hexadecimal values with `0x` (for example, `0xA0FF`).

TABLE 1-1 Sample Literals of Various Data Types

Data Type	Sample Literal
BIGINT	8589934592
INTEGER	186282
SMALLINT	186
NUMERIC	186282.42
DECIMAL	186282.42
DECFLOAT (16)	1234567890123456
REAL	6.02257E23
DOUBLE PRECISION	3.1415926535897E00
FLOAT	6.02257E23
BINARY (2)	'0110011111101010'
VARBINARY (1)	'10011'
CHARACTER(15)	'GREECE '
Note: Fifteen total characters and spaces are between the quote marks above.	
VARCHAR (CHARACTER VARYING)	'lepton'
NATIONAL CHARACTER(15)	'ΕΛΛΑΣ ' [1]
Note: Fifteen total characters and spaces are between the quote marks above.	
NATIONAL CHARACTER VARYING	'λεπτον' [2]
CHARACTER LARGE OBJECT (CLOB)	(A really long character string)
BINARY LARGE OBJECT (BLOB)	(A really long string of ones and zeros)
DATE	DATE '1969-07-20'
TIME(2)	TIME '13.41.32.50'
TIMESTAMP(0)	TIMESTAMP '2007-07-25-13.03.16.000000'
TIME WITH TIMEZONE(4)	TIME '13.41.32.5000-08.00'
TIMESTAMP WITH TIMEZONE(0)	TIMESTAMP '2007-07-25-13.03.16.0000+02.00'
INTERVAL DAY	INTERVAL '7' DAY

[1] *This term is the word that Greeks use to name their own country in their own language. (The English equivalent is Hellas.)*
[2] *This term is the word lepton in Greek national characters.*

Variables vary

Literals, which explicitly hold a single value, are fine if that value appears only once or twice in an application. However, if a value appears multiple times, and if there is any chance that value might change in the future, you should represent it with a variable. That way, if changes are necessary, you have to change the code in one place only, where the value is assigned to the variable, rather than in all the places in the application where that value appears.

For example, suppose an application dealing with a table containing the archives of a magazine retrieves information from various sections of the current issue. One such retrieval might look like this:

```
SELECT Editorial FROM PENGUINLIFE
    WHERE Issue = 47 ;
```

Another could be

```
SELECT LeadStory FROM PENGUINLIFE
    WHERE Issue = 47 ;
```

There could be many more like these two in the application. When next week rolls around and you want to run the application again for the latest issue, you must go through the program by hand and change all the instances of 47 to 48. Computers are supposed to rescue us from such boring, repetitive tasks, and they do. Instead of using literals in such cases, use variables instead, like this:

```
SET @IssueNumber = 48;
SELECT Editorial FROM PENGUINLIFE
    WHERE Issue = @IssueNumber ;
SELECT LeadStory FROM PENGUINLIFE
    WHERE Issue = @IssueNumber ;
```

You have to change the IssueNumber variable in one place only, and the change affects all the places in the application where the variable appears.

Special variables hold specific values

SQL has a few special variables that hold information about system usage. In multiuser systems, you often need to know who is using the system at any given time. This information can be captured in a log file, using the special variables. The special variables are

- SESSION_USER, which holds a value that's equal to the user authorization identifier of the current SQL session. If you write a program that performs a monitoring function, you can interrogate SESSION_USER to find out who is executing SQL statements.
- CURRENT_USER, which stores a user-specified authorization identifier. If a module has no such identifier, CURRENT_USER has the same value as SESSION_USER.
- SYSTEM_USER, which contains the operating system's user identifier. This identifier may differ from that user's identifier in an SQL module. A user may log onto the system as ANDREW, for example, but identify himself to a module as DIRECTOR. The value in SESSION_USER is DIRECTOR. If he makes no explicit specification of the module identifier, and CURRENT_USER also contains DIRECTOR, SYSTEM_USER holds the value ANDREW.

One use of the SYSTEM_USER, SESSION_USER, and CURRENT_USER special variables is to track who is using the system. You can maintain a log table and periodically insert into that table the values that SYSTEM_USER, SESSION_USER, and CURRENT_USER contain. The following example shows how:

```
INSERT INTO USAGELOG (SNAPSHOT)
   VALUES ('User ' || SYSTEM_USER ||
      ' with ID ' || SESSION_USER ||
      ' active at ' || CURRENT_TIMESTAMP) ;
```

This statement produces log entries similar to the following example:

```
User ANDREW with ID DIRECTOR active at 2019-01-03-23.50.00
```


Working with Functions

Functions perform computations or operations that are more elaborate than what you would expect a simple command statement to do. SQL has two kinds of functions: set functions and value functions. *Set functions* are so named because they operate on a set of rows in a table rather than on a single row. *Value functions* operate on the values of fields in a table row.

Summarizing data with set functions

When dealing with a set of table rows, often what you want to know is some aggregate property that applies to the whole set. SQL has five such aggregate or

set functions: COUNT, AVG, MAX, MIN, and SUM. To see how these work, consider the example data for a table named PAPERS in Table 1-2. It is a price table for photographic papers of various sizes and characteristics.

TABLE 1-2 Photographic Paper Price List per 20 Sheets

Paper Type	Size8	Size11
Dual-sided matte	8.49	13.99
Card stock dual-sided matte	9.49	16.95
Professional photo gloss	10.99	19.99
Glossy HW 9M	8.99	13.99
Smooth silk	10.99	19.95
Royal satin	10.99	19.95
Dual-sided semigloss	9.99	17.95
Dual-sided HW semigloss	--	--
Universal two-sided matte	--	--
Transparency	29.95	--

The fields that contain dashes do not have a value. The dash in the table represents a null value.

COUNT

The COUNT function returns the number of rows in a table, or the number of rows that meet a specified condition. In the simplest case, you have

```
SELECT COUNT (*)
   FROM PAPERS ;
```

This returns a value of 10 because there are ten rows in the PAPERS table. You can add a condition to see how many types of paper are available in Size 8:

```
SELECT COUNT (Size8)
   FROM PAPERS ;
```

This returns a value of 8 because, of the ten types of paper in the PAPERS table, only eight are available in size 8. You might also want to know how many different prices there are for papers of size 8. That is also easy to determine:

```
SELECT COUNT (DISTINCT Size8)
   FROM PAPERS ;
```

This returns a value of 6 because there are six distinct values of Size 8 paper. Null values are ignored.

AVG

The `AVG` function calculates and returns the average of the values in the specified column. It works only on columns that contain numeric data.

```
SELECT AVG (Size8)
   FROM PAPERS ;
```

This returns a value of 12.485. If you wonder what the average price is for the Size 11 papers, you can find out this way:

```
SELECT AVG (Size11)
   FROM PAPERS ;
```

This returns a value of 17.539.

MAX

As you might expect, the `MAX` function returns the maximum value found in the specified column. Find the maximum value in the Size8 column:

```
SELECT MAX (Size8)
   FROM PAPERS ;
```

This returns 29.95, the price for 20 sheets of Size 8 transparencies.

MIN

The `MIN` function gives you the minimum value found in the specified column.

```
SELECT MIN (Size8)
   FROM PAPERS ;
```

Here the value returned is 8.49.

SUM

In the case of the photographic paper example, it doesn't make much sense to calculate the sum of all the prices for the papers being offered for sale, but in other applications, this type of calculation can be valuable. Just in case you want to know what it would cost to buy 20 sheets of every Size 11 paper being offered, you could make the following query:

```
SELECT SUM (Size11)
   FROM PAPERS ;
```

It would cost 122.77 to buy 20 sheets of each of the 7 kinds of Size 11 paper that are available.

LISTAGG

LISTAGG is a set function, defined in the SQL:2016 ISO/IEC specification. Its purpose is to transform the values from a group of rows into a list of values delimited by a character that does not occur within the data. An example would be to transform a group of table rows into a string of comma-separated values (CSV).

```
SELECT LISTAGG(LastName, ', ')
         WITHIN GROUP (ORDER BY LastName) "Customer"
  FROM CUSTOMER
  WHERE Zipcode = 97201;
```

This statement will return a list of all customers residing in the 97201 zip code, in ascending order of their last names. This will work as long as there are no commas in the LastName field of any customer.

ANY_VALUE

The ANY_VALUE function, new in SQL:2023, returns any non-null value from the specified data set.

```
SELECT ANY_VALUE(Size8)
    FROM PAPERS;
```

This function will return a randomly selected value from the data values stored in the Size8 column of the data records.

Dissecting data with value functions

A number of data manipulation operations occur fairly frequently. SQL provides value functions to perform these tasks. There are four types of value functions:

- String value functions
- Numeric value functions
- Datetime value functions
- Interval value functions

In the following subsections, I look at the functions available in each of these categories.

String value functions

String value functions take one character string as input and produce another character string as output. There are 12 string value functions:

- `SUBSTRING (FROM)`
- `SUBSTRING (SIMILAR)`
- `UPPER`
- `LOWER`
- `TRIM`
- `LTRIM`
- `RTRIM`
- `LPAD`
- `RPAD`
- `TRANSLATE`
- `CONVERT`
- `OVERLAY`

SUBSTRING (FROM)

The operation of SUBSTRING (FROM) is similar to substring operations in many other computer languages. Here's an example:

```
SUBSTRING ('manual transmission' FROM 8 FOR 4)
```

This returns tran, the substring that starts in the eighth character position and continues for four characters. You want to make sure that the starting point and substring length you specify locate the substring entirely within the source string. If part or all of the substring falls outside the source string, you could receive a result you are not expecting.


REMEMBER

Some implementations do not adhere strictly to the ANSI/ISO standard syntax for the SUBSTRING function, or for the other functions that follow. Check the documentation of the implementation you are using if the code samples given here do not work for you.

SUBSTRING (SIMILAR)

SUBSTRING (SIMILAR) is a regular expression substring function. It divides a string into three parts and returns the middle part. Formally, a regular expression is a string of legal characters. A substring is a particular designated part of that string. Consider this example:

```
SUBSTRING ('antidisestablishmentarianism'
           SIMILAR 'antidis\"[:ALPHA:]+\"arianism'
           ESCAPE '\' )
```

The original string is the first operand. The operand following the SIMILAR keyword is a character string literal that includes a regular expression in the form of another character string literal, a separator (\"), a second regular expression that means "one or more alphabetic characters," a second separator (\"), and a third regular expression in the form of a different character string literal. The value returned is

```
establishment
```

UPPER

The UPPER function converts its target string to all uppercase.

```
UPPER ('ChAoTic')            returns 'CHAOTIC'
```

The UPPER function has no effect on character sets, such as Hebrew, that do not distinguish between upper- and lowercase.

LOWER

The LOWER function converts its target string to all lowercase.

```
LOWER ('INTRUDER ALERT!')      returns 'intruder alert!'
```

As is the case for UPPER, LOWER has no effect on character sets that do not include the concept of case.

TRIM

In the SQL:2023 standard, the TRIM function enables you to crop a string, shaving off one character at the front or the back of the string — or both. Here are a few examples:

```
TRIM (LEADING ' ' FROM ' ALERT ')     returns 'ALERT '
TRIM (TRAILING ' ' FROM ' ALERT ')    returns ' ALERT'
TRIM (BOTH ' ' FROM ' ALERT ')        returns 'ALERT'
TRIM (LEADING 'A' FROM 'ALERT')       returns 'LERT'
```

If you don't specify what to trim, the blank space ('') is the default.

WARNING

Many current SQL implementations don't follow this standard behavior for the TRIM function. Instead, the TRIM function they implement behaves similar to the new LTRIM and RTRIM functions (described in the following section).

LTRIM AND RTRIM

Because of the disparate behavior of the TRIM function implemented by various database packages, the SQL:2023 standard tries to accommodate the different implementations by adding the LTRIM and RTRIM functions. These functions expand on the standard TRIM function by allowing you to shave off multiple instances of the specified character left (LTRIM) or right (RTRIM) of the specified string.

```
LTRIM('    ALERT ', ' ')    returns 'ALERT '
RTRIM(' ALERT    ', ' ')    returns ' ALERT'
```

As with the TRIM function, if you don't specify what to trim, the blank space (' ') is the default.

TIP

Unfortunately, there is no function for trimming multiple characters off of both the left *and* right sides of the string. To do that, you'll need to use both functions on the same string.

LPAD AND RPAD

The SQL:2023 standard adds the `LPAD` and `RPAD` functions, which allow you to pad a string value to a specific size.

```
LPAD('Rich', 10, ' ')    returns '      Rich'
RPAD('Rich', 10, ' ')    returns 'Rich      '
```

As with the `TRIM` function, if you don't specify what to pad, the padding character is the blank space (`' '`) by default.

TRANSLATE AND CONVERT

The `TRANSLATE` and `CONVERT` functions take a source string in one character set and transform the original string into a string in another character set. Examples might be Greek to English or Katakana to Norwegian. The conversion functions that specify these transformations are implementation-specific, so I don't give any details here.

These functions do not really translate character strings from one language to another. All they do is translate a character from the first character set to the corresponding character in the second character set. In going from Greek to English, it would convert `Ελλασ` to `Ellas` instead of translating it as `Greece`. ("`Ελλασ`" is what the Greeks call their country. I have no idea why English speakers call it Greece.)

OVERLAY

The `OVERLAY` function is a `SUBSTRING` function with a little extra functionality. As with `SUBSTRING`, it finds a specified substring within a target string. However, instead of returning the string that it finds, it replaces it with a different string. For example:

```
OVERLAY ('I Love Paris' PLACING 'Tokyo' FROM 8 FOR 5)
```

This changes the string to

```
I Love Tokyo
```

This won't work if you want to change I Love Paris to I Love London. The number of letters in London does not match the number in Paris.

Numeric value functions

Numeric value functions can take a variety of data types as input, but the output is always a numeric value. SQL has 14 types of numeric value functions. The defining characteristic of a function is that it returns a value of some sort. Numeric value functions always return a numeric value. Thus, the square root function will return a value that is the square root of the input; the natural logarithm function will return a value that is the natural logarithm of the input, and so on.

- Position expression (`POSITION`)
- Extract expression (`EXTRACT`)
- Length expression (`CHAR_LENGTH`, `CHARACTER_LENGTH`, `OCTET_LENGTH`)
- Cardinality expression (`CARDINALITY`)
- Absolute value expression (`ABS`)
- Modulus expression (`MOD`)
- Trigonometric functions (`SIN`, `COS`, `TAN`, `ASIN`, `ACOS`, `ATAN`, `SINH`, `COSH`, `TANH`)
- Logarithmic functions (`LOG`, `LOG10`, `LN`)
- Exponential function (`EXP`)
- Power function (`POWER`)
- Square root (`SQRT`)
- Floor function (`FLOOR`)
- Ceiling function (`CEIL`, `CEILING`)
- Greatest function (`GREATEST`)
- Least function (`LEAST`)
- Width bucket function (`WIDTH_BUCKET`)

POSITION

`POSITION` searches for a specified target string within a specified source string and returns the character position where the target string begins. The syntax is as follows:

```
POSITION (target IN source)
```

Table 1-3 shows a few examples.

TABLE 1-3 Sample Uses of the POSITION Statement

This Statement	Returns
`POSITION ('T' IN 'Transmission, automatic')`	1
`POSITION ('Tra' IN 'Transmission, automatic')`	1
`POSITION ('au' IN 'Transmission, automatic')`	15
`POSITION ('man' IN 'Transmission, automatic')`	0
`POSITION ('' IN 'Transmission, automatic')`	1

If the function doesn't find the target string, the `POSITION` function returns a zero value. If the target string has zero length (as in the last example), the `POSITION` function always returns a value of 1. If any operand in the function has a null value, the result is a null value.

EXTRACT

The `EXTRACT` function extracts a single field from a datetime or an interval. The following statement, for example, returns 12:

```
EXTRACT (MONTH FROM DATE '2018-12-04')
```

CHARACTER_LENGTH

The `CHARACTER_LENGTH` function returns the number of characters in a character string. The following statement, for example, returns 20:

```
CHARACTER_LENGTH ('Transmission, manual')
```

REMEMBER

As you can see, commas and even blank spaces count as characters. Note that this function is not particularly useful if its argument is a literal like `'Transmission, manual'`. I can write `20` just as easily as I can write `CHARACTER_LENGTH ('Transmission, manual')`. In fact, writing `20` is easier. This function is more useful if its argument is an expression rather than a literal value.

OCTET_LENGTH

In music, a vocal ensemble made up of eight singers is called an *octet*. Typically, the parts that the ensemble represents are first and second soprano, first and

second alto, first and second tenor, and first and second bass. In computer terminology, an ensemble of eight data bits is called a *byte*. The word *byte* is clever in that the term clearly relates to *bit* but implies something larger than a bit. A nice wordplay — but unfortunately, nothing in the word *byte* conveys the concept of "eightness." By borrowing the musical term, a more apt description of a collection of eight bits becomes possible.

Practically all modern computers use eight bits to represent a single alphanumeric character. More complex character sets (such as Chinese) require 16 bits to represent a single character. The `OCTET_LENGTH` function counts and returns the number of octets (bytes) in a string. If the string is a bit string, `OCTET_LENGTH` returns the number of octets you need to hold that number of bits. If the string is an English-language character string (with one octet per character), the function returns the number of characters in the string. If the string is a Chinese character string, the function returns a number that is twice the number of Chinese characters. The following string is an example:

```
OCTET_LENGTH ('Brakes, disc')
```

This function returns 12 because each character takes up one octet.

Some character sets use a variable number of octets for different characters. In particular, some character sets that support mixtures of Kanji and Latin characters use *escape* characters to switch between the two character sets. A string that contains both Latin and Kanji may have, for example, 30 characters and require 30 octets if all the characters are Latin; 62 characters if all the characters are Kanji (60 characters plus a leading and trailing shift character); and 150 characters if the characters alternate between Latin and Kanji (because each Kanji character needs two octets for the character and one octet each for the leading and trailing shift characters). The `OCTET_LENGTH` function returns the number of octets you need for the current value of the string.

CARDINALITY

Cardinality deals with collections of elements such as arrays or multisets, where each element is a value of some data type. The cardinality of the collection is the number of elements that it contains. One use of the `CARDINALITY` function is something like this:

```
CARDINALITY (TeamRoster)
```

This function would return 12, for example, if there were 12 team members on the roster. `TeamRoster`, a column in the TEAM table, can be either an array or a multiset. An *array* is an ordered collection of elements, and a *multiset* is an unordered

collection of elements. For a team roster, which changes frequently, a multiset makes more sense. (You can find out more about arrays and multisets in Book 1, Chapter 6.)

ABS

The `ABS` function returns the absolute value of a numeric value expression.

```
ABS (–273)
```

This returns 273.

TRIGONOMETRIC FUNCTIONS SIN, COS, TAN, ASIN, ACOS, ATAN, SINH, COSH, TANH

The trig functions give you the values you would expect, such as the sine of an angle or the hyperbolic tangent of one.

LOGARITHMIC FUNCTIONS LOG10, LN, LOG (<BASE>, <VALUE>)

The logarithmic functions enable you to generate the logarithm of a number, either a base-10 logarithm, a natural logarithm, or a logarithm to a base that you specify.

MOD

The `MOD` function returns the *modulus* — the remainder of division of one number by another — of two numeric value expressions.

```
MOD (6,4)
```

This function returns 2, the modulus of six divided by four.

EXP

This function raises the base of the natural logarithms *e* to the power specified by a numeric value expression:

```
EXP (2)
```

This function returns something like 7.389056. The number of digits beyond the decimal point is implementation-dependent.

POWER

This function raises the value of the first numeric value expression to the power of the second numeric value expression:

```
POWER (3,7)
```

This function returns 2187, which is three raised to the seventh power.

SQRT

This function returns the square root of the value of the numeric value expression:

```
SQRT (9)
```

This function returns 3, the square root of nine.

FLOOR

This function rounds the numeric value expression to the largest integer not greater than the expression:

```
FLOOR (2.73)
```

This function returns 2.0.

CEIL OR CEILING

This function rounds the numeric value expression to the smallest integer not less than the expression.

```
CEIL (2.73)
```

This function returns 3.0.

GREATEST

Because the set functions already use the `MAX` function, the SQL:2023 standard defines the `GREATEST` function to use for numeric values.

```
GREATEST(1.0,2.0,3.5)
```

This functions returns 3.5.

LEAST

Because the set functions already use the MIN function, the SQL:2023 standard defines the LEAST function to use for numeric values.

```
LEAST(1.0, 2.0, 3.5)
```

This function returns 1.0.

WIDTH_BUCKET

The WIDTH_BUCKET function, used in online application processing (OLAP), is a function of four arguments, returning an integer between the value of the second (minimum) argument and the value of the third (maximum) argument. It assigns the first argument to an equiwidth partitioning of the range of numbers between the second and third arguments. Values outside this range are assigned to either the value of zero or one more than the fourth argument (the number of buckets).

For example:

```
WIDTH_BUCKET (PI, 0, 10, 5)
```

Suppose PI is a numeric value expression with a value of 3.141592. The example partitions the interval from zero to ten into five equal *buckets*, each with a width of two. The function returns a value of 2 because 3.141592 falls into the second bucket, which covers the range from two to four.

Datetime value functions

SQL includes three functions that return information about the current date, current time, or both. CURRENT_DATE returns the current date; CURRENT_TIME returns the current time; and CURRENT_TIMESTAMP returns both the current date and the current time. CURRENT_DATE doesn't take an argument, but CURRENT_TIME and CURRENT_TIMESTAMP both take a single argument. The argument specifies the precision for the seconds part of the time value that the function returns. Datetime data types and the precision concept are described in Book 1, Chapter 6.

The following table offers some examples of these datetime value functions.

This Statement	Returns
CURRENT_DATE	2019-01-23
CURRENT_TIME (1)	08:36:57.3
CURRENT_IMESTAMP (2)	2019-01-23 08:36:57.38

The date that CURRENT_DATE returns is DATE type data. The time that CURRENT_TIME (*p*) returns is TIME type data, and the timestamp that CURRENT_TIMESTAMP (*p*) returns is TIMESTAMP type data. The precision (*p*) specified is the number of digits beyond the decimal point, showing fractions of a second. Because SQL retrieves date and time information from your computer's system clock, the information is correct for the time zone in which the computer resides.

In some applications, you may want to deal with dates, times, or timestamps as character strings to take advantage of the functions that operate on character data. You can perform a type conversion by using the CAST expression, which I describe later in this chapter.

Polymorphic table functions

A table function is a user-defined function that returns a table as a result. A polymorphic table function, first described in SQL:2016, is a table function whose row type is not declared when the function is created. Instead, the row type may depend on the function arguments used when the function is invoked.

Using Expressions

An *expression* is any combination of elements that reduces to a single value. The elements can be numbers, strings, dates, times, intervals, Booleans, or more complex things. What they are doesn't matter, as long as after all operations have taken place, the result is a single value.

Numeric value expressions

The operands in a numeric value expression can be numbers of an exact numeric type or of an approximate numeric type. (Exact and approximate numeric types are discussed in Book 1, Chapter 6.) Operands of different types can be used within a single expression. If at least one operand is of an approximate type, the result is of an approximate type. If all operands are of exact types, the result is of an exact type. The SQL specification does not specify exactly what type the result of any given expression will be, due to the wide variety of platforms that SQL runs on.

Here are some examples of valid numeric value expressions:

- -24
- 13+78

- 4*(5+8)
- Weight/(Length*Width*Height)
- Miles/5280

String value expressions

String value expressions can consist of a single string or a concatenation of strings. The concatenation operator (||) joins two strings together and is the only one you can use in a string value expression. Table 1-4 shows some examples of string value expressions and the strings that they produce.

TABLE 1-4 Examples of String Value Expressions

String Value Expression	Resulting String
`'nanotechnology'`	`'nanotechnology'`
`'nano' || 'technology'`	`'nanotechnology'`
`'nano' || '' || 'technology'`	`'nanotechnology'`
`'Isaac' || '' || 'Newton'`	`'Isaac Newton'`
`FirstName || ' ' || LastName`	`'Isaac Newton'`
`B'10101010' || B'01010101'`	`B'1010101001010101'`

From the first two rows in Table 1-4, you see that concatenating two strings produces a result string that has seamlessly joined the two original strings. The third row shows that concatenating a null value with two source strings produces the same result as if the null were not there. The fourth row shows concatenation of two strings while retaining a blank space in between. The fifth row shows the concatenation of two variables with a blank space in between produces a string consisting of the values of those variables separated by a blank space. Finally, the last line of Table 1-4 shows the concatenation of two binary strings. The result is a single binary string that is a seamless combination of the two source strings.

Datetime value expressions

Datetime value expressions perform operations on dates and times. Such data is of the `DATE`, `TIME`, `TIMESTAMP`, or `INTERVAL` type. The result of a datetime value expression is always of the `DATE`, `TIME`, or `TIMESTAMP` type. Intervals are not one of

the datetime types, but an interval can be added to or subtracted from a datetime to produce another datetime. Here's an example datetime value expression that makes use of an added interval:

```
CURRENT_DATE + INTERVAL '2' DAY
```

This expression evaluates to the day after tomorrow.

Datetimes can also include time zone information. The system maintains times in Coordinated Universal Time (UTC), which until recently was known as Greenwich Mean Time (GMT). (I guess the feeling was that Greenwich was too provincial, and a more general name for world time was called for.) You can specify a time as being either at your local time, or as an offset from UTC. An example is

```
TIME '13:15:00' AT LOCAL
```

for 1:15 p.m. local time. Another example is

```
TIME '13:15:00' AT TIME ZONE INTERVAL '-8:00' HOUR TO MINUTE
```

for 1:15 p.m. Pacific Standard Time. (Pacific Standard Time is eight hours earlier than UTC.)

Interval value expressions

An *interval* is the difference between two datetimes. If you subtract one datetime from another, the result is an interval. It makes no sense to add two datetimes, so SQL does not allow you to do it.

There are two kinds of intervals: year-month and day-time. This situation is a little messy, but necessary because not all months contain the same number of days. Because a month can be 28, 29, 30, or 31 days long, there is no direct translation from days to months. As a result, when using an interval, you must specify which kind of interval it is. Suppose you expect to take an around-the-world cruise after you retire, starting on June 1, 2045. How many years and months is that from now? An interval value expression gives you the answer.

```
(DATE '2045-06-01' - CURRENT_DATE) YEAR TO MONTH
```

You can add two intervals to obtain an interval result.

```
INTERVAL '30' DAY + INTERVAL '14' DAY
```

However, you cannot do the following:

```
INTERVAL '30' DAY + INTERVAL '14' MONTH
```

The two kinds of intervals do not mix. Besides addition and subtraction, multiplication and division of intervals also are allowed. The expression

```
INTERVAL '7' DAY * 3
```

is valid and gives an interval of 21 days. The expression

```
INTERVAL '12' MONTH / 2
```

is also valid and gives an interval of 6 months. Intervals can also be negative.

```
INTERVAL '-3' DAY
```

gives an interval of -3 days. Aside from the literals I use in the previous examples, any value expression or combination of value expressions that evaluates to an interval can be used in an interval value expression.

Boolean value expressions

Only three legal Boolean values exist: `TRUE`, `FALSE`, and `UNKNOWN`. The `UNKNOWN` value becomes operative when a `NULL` is involved. Suppose the Boolean variable `Signal1` is `TRUE` and the Boolean variable `Signal2` is `FALSE`. The following Boolean value expression evaluates to `TRUE`:

```
Signal1 IS TRUE
```

So does this one:

```
Signal1 IS TRUE OR Signal2 IS TRUE
```

However, the following Boolean value expression evaluates to `FALSE`.

```
Signal1 IS TRUE AND Signal2 IS TRUE
```

The `AND` operator means that both predicates must be true for the result to be true. (A *predicate* is an expression that asserts a fact about values.) Because `Signal2` is false, the entire expression evaluates to a `FALSE` value.

Array value expressions

You can use a couple of types of expressions with arrays. The first has to do with cardinality. The maximum number of elements an array can have is called the array's *maximum cardinality*. The actual number of elements in the array at a given time is called its *actual cardinality*. You can combine two arrays by concatenating them, summing their maximum cardinalities in the process. Suppose you want to know the actual cardinality of the concatenation of two array-type columns in a table, where the first element of the first column has a given value. You can execute the following statement:

```
SELECT CARDINALITY (FirstColumn || SecondColumn)
   FROM TARGETTABLE
WHERE FirstColumn[1] = 42 ;
```

The `CARDINALITY` function gives the combined cardinality of the two arrays, where the first element in the first array has a value of 42.

Note: The first element of an SQL array is considered to be element 1, rather than element 0 as is true for some other languages.

Conditional value expressions

The value of a conditional value expression depends on a condition. SQL offers three variants of conditional value expressions: `CASE`, `NULLIF`, and `COALESCE`. I look at each of these separately.

Handling different cases

The `CASE` conditional expression was added to SQL to give it some of the functionality that all full-featured computer languages have, the ability to do one thing if a condition holds and another thing if the condition does not hold. Originally conceived as a data sublanguage that was concerned only with managing data, SQL has gradually gained features that enable it to take on more of the functions needed by application programs.

SQL actually has two different `CASE` structures: the `CASE` expression described here, and a `CASE` statement. The `CASE` expression, like all expressions, evaluates to a single value. You can use a `CASE` expression anywhere where a value is legal. The `CASE` statement, on the other hand, doesn't evaluate to a value. Instead, it executes a block of statements.

The `CASE` expression searches a table, one row at a time, taking on the value of a specified result whenever one of a list of conditions is `TRUE`. If the first condition

is not satisfied for a row, the second condition is tested, and if it is TRUE, the result specified for it is given to the expression, and so on until all conditions are processed. If no match is found, the expression takes on a NULL value. Processing then moves to the next row.

SEARCHING FOR TABLE ROWS THAT SATISFY VARIOUS CONDITIONS

You can specify the value to be given to a CASE expression, based on which of several conditions is satisfied. Here's the syntax:

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    WHEN conditionN THEN resultN
    ELSE resultx
END
```

If, in searching a table, the CASE expression finds a row where condition1 is true, it takes on the value of result1. If condition1 is not true, but condition2 is true, it takes on the value of result2. This continues for all conditions. If none of the conditions are met and there is no ELSE clause, the expression is given the NULL value. Here's an example of usage:

```
UPDATE MECHANIC
    Set JobTitle = CASE
                       WHEN Specialty = 'Brakes'
                              THEN 'Brake Fixer'
                       WHEN Specialty = 'Engines'
                              THEN 'Motor Master'
                       WHEN Specialty = 'Electrical'
                              THEN 'Wizard'
                       ELSE 'Apprentice'
                   END ;
```

THE EQUALITY CONDITION ALLOWS A COMPACT CASE VALUE EXPRESSION

A shorthand version of the CASE statement can be used when the condition, as in the previous example, is based on one thing being equal (=) to one other thing. The syntax is as follows:

```
CASE valuet
```

```
    WHEN value1 THEN result1
    WHEN value2 THEN result2
    ...
    WHEN valueN THEN resultN
    ELSE resultx
END
```

For the preceding example, this translates to

```
UPDATE MECHANIC
    Set JobTitle = CASE Specialty
                       WHEN 'Brakes' THEN 'Brake Fixer'
                       WHEN 'Engines' THEN 'Motor Master'
                       WHEN 'Electrical' THEN 'Wizard'
                       ELSE 'Apprentice'
                   END ;
```

If the condition involves anything other than equality, the first, nonabbreviated form must be used.

The NULLIF special CASE

SQL databases are unusual in that `NULL` values are allowed. A `NULL` value can represent an unknown value, a known value that has just not been entered into the database yet, or a value that does not exist. Most other languages that deal with data do not support nulls, so whenever a situation arises in such databases where a value is not known, not yet entered, or nonexistent, the space is filled with a value that would not otherwise occur, such as `-1` in a field that never holds a negative value, or `***` in a character field in which asterisks are not valid characters.

To migrate data from a database that does not support nulls to an SQL database that does, you can use a `CASE` statement such as

```
UPDATE MECHANIC
    SET Specialty = CASE Specialty
                       WHEN '***' THEN NULL
                       ELSE Specialty
                    END ;
```

You can do the same thing in a shorthand manner, using a `NULLIF` expression, as follows:

```
UPDATE MECHANIC
   SET Specialty = NULLIF(Specialty, '***') ;
```

Admittedly, this looks more cryptic than the CASE version, but it does save some tedious typing. You could interpret it as, "Update the MECHANIC table by setting the value of Specialty to NULL if its current value is '***'".

Bypassing null values with COALESCE

The COALESCE expression is another shorthand version of CASE that deals with NULL values. It examines a series of values in a table row and assumes the value of the first one that is not NULL. If all the listed values are NULL, the COALESCE expression takes on the NULL value. Here's the syntax for a CASE expression that does this:

```
CASE
    WHEN value1 IS NOT NULL
         THEN value1
    WHEN value2 IS NOT NULL
         THEN value2
    ...
    WHEN valueN is NOT NULL
         THEN valueN
    ELSE NULL
END
```

Here's the syntax for the equivalent COALESCE expression:

```
COALESCE(value1, value2, ..., valueN)
```

If you are dealing with a large number of cases, the COALESCE version can save you quite a bit of typing.

Converting data types with a CAST expression

In Book 1, Chapter 6, I describe the data types that SQL recognizes. The host languages that SQL statements are often embedded in also recognize data types, and those host language data types are never an exact match for the SQL data types. This could present a problem, except for the fact that, with a CAST expression, you can convert data of one type into data of another type. Whereas the first type might not be compatible with the place you want to send the data, the second type is. Of course, not all conversions are possible. If you have a character string such as '2019-02-14', you can convert it to the DATE type with a CAST expression. However, SQL doesn't let you convert a character string such

as 'rhinoceros' to the DATE type. The data to be converted must be compatible with the destination type.

Casting one SQL data type to another

The simplest kind of cast is from one SQL data type to another SQL data type. Even for this operation, however, you cannot indiscriminately make any conversion you want. The data you are converting must be compatible with the target data type. For example, suppose you have a table named ENGINEERS with a column named SSN, which is of the NUMERIC type. Perhaps you have another table, named MANAGERS, that has a column named SocSecNo, which is of the CHAR (9) type. A typical entry in SSN might be 987654321. To find all the engineers who are also managers, you can use the following query. The CAST expression converts the CHAR (9) type to the NUMERIC type so that the operation can proceed.

```
SELECT * FROM ENGINEER
    WHERE ENGINEER.SSN = CAST(MANAGER.SocSecNo AS INTEGER) ;
```

This returns all the rows from the ENGINEER table that have Social Security Numbers that match Social Security Numbers in the MANAGERS table. To do so, it changes the Social Security Number from the MANAGER table from the CHAR (9) type to the INTEGER type, for the purposes of the comparison.

Using CAST to overcome data type incompatibilities between SQL and its host language

Problems arise when you want to send data between SQL and its host language. For example, SQL has the DECIMAL and NUMERIC types, but some host languages, such as FORTRAN and Pascal, do not. One way around this problem is to use CAST to put a numeric value into a character string, and then put the character string into a host variable that the host language can take in and deal with.

Suppose you maintain salary information as REAL type data in the EMPLOYEE table. You want to make some manipulations on that data that SQL is not well-equipped to perform, but your host language is. You can cast the data into a form the host language can accept, operate on it at the host level, and then cast the result back to a form acceptable to the SQL database.

```
SELECT CAST(Salary AS CHAR (10)) INTO :salary_var
    FROM EMPLOYEE
    WHERE EmpID = :emp_id_var ;
```

That puts the salary value where the host language can grab it, and in a form that the host language understands. After the host language is finished operating on the data item, it can return to the SQL database via a similar path:

```
UPDATE EMPLOYEE
    SET Salary = CAST(:salary_var AS DECIMAL(10,2))
         WHERE EmpID = :emp_id_var ;
```

In addition to these conversions, you can do a number of other conversions, including the following:

- Any numeric type to any other numeric type
- Any exact numeric type to a single-component interval, such as `INTERVAL DAY`
- Any `DATE` to a `TIMESTAMP`
- Any `TIME` to a `TIME` with a different fractional seconds precision or a `TIMESTAMP`
- Any `TIMESTAMP` to a `DATE`, a `TIME`, or a `TIMESTAMP` with a different fractional seconds precision
- Any year-month `INTERVAL` to an exact numeric type
- Any day-time `INTERVAL` to an exact numeric type
- Any character string to any other type, where the data makes sense
- Any bit string to a character string
- A Boolean to a character string

Row value expressions

Row value expressions (as distinct from mere row values, which are covered at the beginning of this chapter) enable you to deal with the data in an entire table row or a subset of a row. The other expressions that I've shown deal only with a single field in a row at a time. Row value expressions are useful for adding new data to a table a row at a time, or to specify the retrieval of multiple fields from a table row. Here's an example of a row value expression used to enter a new row of data to a table:

```
INSERT INTO CERTIFICATIONS
   (CertificationNo, CertName, MechanicID, Expires)
   VALUES
    (1, 'V8 Engines', 34, 2021-07-31) ;
```

One advantage of using row value expressions is that many SQL implementations can process them faster than the equivalent one-field-at-a-time operations. This could make a significant difference in performance at runtime.