2

Built-In Functions

As we've already seen, SQL is a powerful tool for manipulating data. Part of what makes the language so powerful is the built-in functions (BIFs). These allow you to perform complex tasks with a single statement within an SQL program. This capability becomes even more powerful when you consider that you can use these functions and procedures within any of the embedded SQL languages or within DB2 Query Manager for creating printed reports.

In this chapter, we'll explore the BIFs included in the DB2 implementation of SQL on the iSeries. Before we begin, however, it's important to understand what the similarities and differences are between a *function* and a *procedure*. These terms are used with SQL in much the same way that they would be used within any other programming language, including ILE RPG. A *function* is a compiled set of statements that performs a required task and returns a value. Functions can be used to perform simple tasks, like calculating or manipulating a value, or more complex tasks, like looking up a value in a database table and returning that value to the caller. A *procedure* is a compiled set of statements that performs a required task but does not return a value. Generally, an SQL *stored procedure*

is simply a compiled program that can be called through an SQL call. This gives us the ability to use a compiled program like this from client applications through ADO, ODBC, or JDBC. Let's start by examining the BIFs in DB2 on the iSeries.

Built-In Functions

The SQL BIFs in DB2 can be broken down into two main categories. *Columnar* SQL functions are used to calculate summary-level values. *Scalar* SQL functions are used at a detail level. Since the uses of these two types of functions are distinctly different, we'll examine each group of functions separately.

Columnar Functions

These functions allow you to create total- or subtotal-level summaries. Subtotal-level groupings are defined using the GROUP BY clause. Table 2.1 displays a list of the columnar functions supported in DB2 along with a brief description of their use. This group of functions allows you to summarize data sets.

Table 2.1: DB2 Columnar Functions

Function	Description
AVG	This function returns an average of the field or value on the supplied parameter over the defined data set.
SUM	This function returns the total of the supplied field values within the defined data set.
COUNT	This function determines the number of rows within the defined data set.
COUNT_BIG	This function also returns the number of rows within the defined data set, but it supports more digits than the standard integer value returned by the COUNT function.
MAX	This function returns the maximum value for the field supplied on its parameter within the defined data set.

Table 2.1: DB2 Columnar Functions (continued)

Function	Description
MIN	This function is the opposite of the MAX function. It returns the minimum value within the defined data set.
STDDEV_POP or STDEV	These functions return the standard deviation (or the square root of the variance) of the supplied field value within the defined data set.
VAR_POP or VARIANCE or VAR	These functions return the variance calculated on the supplied field value within the defined data set.

The AVG function calculates the average value of a field within the data set. The following example shows how to use this function to calculate an average item price from an order file.

```
SELECT ITEM, AVG(PRICE)
FROM ORDERDTL
GROUP BY ITEM
```

The GROUP BY clause defines the level at which the values are summarized.

The SUM function performs much as you would assume. It totals the values from the specified field or fields. With many of the columnar functions, it's possible to use multiple fields or values. The example below illustrates this by using the SUM function to calculate the total dollar value of orders from the same order file used in the previous example.

```
SELECT ORDNO, SUM(PRICE * QTY)
FROM ORDERDTL
GROUP BY ORDNO
```

When this example is executed, the value of PRICE * QTY is calculated and then used as our summary field.

The COUNT function can be used to count the number of rows in a group of records within a data set. If we add the COUNT function to the prior example, we can determine the number of order lines that make up each order.

```
SELECT ORDNO, SUM(PRICE * QTY) AS VALUE, COUNT(*) AS LINES
FROM ORDERDTL
GROUP BY ORDNO
HAVING SUM(PRICE * QTY)>0
```

Within this example, you'll notice that we've added the AS modifier to name the field created by each of our functions. You'll also notice that we've added the HAVING clause. It is acceptable to include a columnar function in the HAVING clause of a SELECT statement. In this example, groups will be displayed only if the SUM(PRICE * QTY) is greater than 0.

The COUNT_BIG function works exactly like the COUNT function, but while the COUNT function can only return a value of up to 15 digits, the COUNT_BIG function supports a value of up to 31 digits.

The MIN and MAX functions return the minimum and maximum values for the specified field in a group within a data set. The following example uses these two functions to determine the earliest and latest dates that an item was received.

```
SELECT ITEM, MIN(ORDDATE), MAX(ORDDATE)
FROM RECEIPTS
GROUP BY ITEM
```

While each of these functions is based on the same field, the values returned by the functions can be distinctly different. This statement would return both the earliest and the most recent order date value for each item in the file RECEIPTS.

The functions VAR (or VAR_POP or VARIANCE) and STDDEV (or STDEV_POP) are used for statistical analysis. A variance is calculated by taking the mean of the square root of the difference between the mean value in a series of numbers and each number itself. Figure 2.1 illustrates calculating a variance on the numbers 1, 4, and 13.

$$variance = \frac{(1-6)^2 + (4-6)^2 + (13-6)^2}{3}$$

Figure 2.1: This simple illustration shows the formula to calculate a variance.

In this example, the mean or average value in our series is 6. The difference between the values and 6 is calculated, giving us -5, -2, and 7, respectively. These values are then squared, which results in 25, 4, and 49, respectively. The sum of these values (78) is then divided by the number of values (3), which gives us a variance of 26. The standard deviation is simply the square root of the variance. Using our previous example, we would arrive at a standard deviation of 5.09902. Generally, these two functions are used to give an idea of the range of values that exist within a set of numbers. The different versions of the functions are included for maximum compatibility with other SQL implementations.

As we've seen here, columnar functions give us a way to take volumes of data and summarize them into a form that is more easily analyzed. Next, we'll go through the scalar functions available.

Scalar Functions

Scalar functions allow us to extend existing values within a data set. These functions can be used not only as part of the SELECT list and HAVING clauses, but also as part of file join and WHERE conditions. In addition, they can be used as a part of a GROUP BY or ORDER BY clause. Basically, anywhere that a field name or value can be specified, a scalar function can be used.

DB2 UDB for the iSeries supports more than 150 scalar functions that allow you to control any type of field—from numeric to string to date. Some of these functions will be familiar to you because of their similarity to functions in other common languages. Table 2.2 breaks down these functions into groups based on their use.

Table 2.2: DB2 Scalar Functions

Numeric	String	Date/Time	Logical
ABS	BINARY*	CURDATE	LAND
ACOS	BIT_LENGTH*	CURTIME	LNOT
ANTILOG	CHAR	DATE	LOR
ASIN	CHARACTER_LENGTH	DAY	XOR
ATAN	CHAR_LENGTH	DAYNAME*	
ATANH	CONCAT	DAYOFMONTH	**Other**
ATAN2	DIFFERENCE	DAYOFWEEK	BLOB
BIGINT	DIGITS	DAYOFWEEK_ ISO	CLOB
CEILING	GRAPHIC	DAYOFYEAR	COALESCE
COS	INSERT*	DAYS	DATABASE*
COSH	LCASE	EXTRACT*	DATAPARTITIONNAME*
COT	LEFT	HOUR	DATAPARTITIONNUM*
DECIMAL	LENGTH	JULIAN_DAY	DBCLOB
DEGREES	LOCATE	MICROSECOND	DBPARTITIONNAME*
DOUBLE	LOWER	MIDNIGHT_ SECONDS	DBPARTITIONNUM*
DOUBLE_PRECISION	LTRIM	MINUTE	DECRYPT_ BINARY*
EXP	OCTET_LENGTH*	MONTH	DECRYPT_ BIT*
FLOAT	POSITION	MONTHNAME*	DECRYPT_CHAR*
FLOOR	POSSTR	NOW	DECRYPT_DB*
INT	REPEAT*	QUARTER	ENCRYPT_RC2*
INTEGER	REPLACE*	SECOND	GETHINT*
LN	RIGHT*	TIME	HASH
LOG10	RTRIM	TIMESTAMP	HASHED_VALUE*

Table 2.2: DB2 Scalar Functions (continued)

Numeric	String	Date/Time	Logical
MOD	SOUNDEX	TIMESTAMP_ISO*	HEX
MULTIPLY_ALT*	SPACE	TIMESTAMPDIFF	IDENTITY_VAL_LOCAL
PI	STRIP	WEEK	IFNULL
POWER	SUBST	WEEK_ISO	MAX
RADIANS	SUBSTRING	YEAR	MIN
RAND	TRANSLATE		NODENAME
REAL	TRIM	**URL**	NULL
REPEAT	UCASE	DLCOMMENT	NULLIF
REPLACE	UPPER	DLLINKTYPE	RRN
ROUND	VARBINARY*	DLURLCOMPLETE	VALUE
SIGN	VARCHAR	DLURLPATH	
SIN	VARGRAPHIC	DLURLPATHONLY	
SINH		DLURLSCHEME	
SMALLINT		DLURLSERVER	
SQRT		DLVALUE	
STRIP			
TAN			
TANH			
TRUNC			
TRUNCATE			
ZONED			

Note: Functions noted by an asterisk (*) indicate new functions added between V5R2 and V5R3.

Let's review the groups of functions one at a time. The first group of functions focuses on numeric operations. You'll probably find that you use these functions more than the other groups.

Numeric Functions

As the grouping suggests, these functions perform operations on numeric fields. The ABS function returns the absolute value of the provided field or value. The following statement is an example of using ABS.

```
SELECT ABS(TRANQT)
       FROM TRANSFILE
```

Assuming the value of the field TRANQT is -25, the value returned would be 25.

The ACOS function converts the supplied value to an arc cosine value. This function performs the opposite operation to the COS function. The value returned is the angle in radians. The following statement shows a sample of the ACOS function.

```
SELECT ACOS(.25)
       FROM MYFILE
```

Using this sample statement, the ACOS function would return 1.318116072.

The COS function returns the cosine value for the provided value. As with the ACOS function, the value returned is an angle in radians. If in the previous example we replaced the ACOS function with the COS function, the value returned would be 0.968912422.

ANTILOG returns the base 10 anti-logarithm value for the provided value. This function performs the opposite operation of the LOG function. When executed, the ANTILOG function will evaluate 10^*x* where *x* is the supplied value. The following example calculates the ANTILOG and LOG for the number 3.

```
SELECT ANTILOG(3), LOG(3)
       FROM MYFILE
```

When executed, this function returns 1.0000000000000007E+003 and 4.7712125471966244E-001, respectively. These values converted to decimal would evaluate to 1,000 and .447, respectively.

The ASIN function calculates the arc sine of a given number. This and the SIN function, which calculates the sine of a given number, are opposites to each other. When I refer to two functions as opposites, what I am saying is that the ASIN(SIN(*x*)) will equal *x* and the SIN(ASIN(*y*)) will equal *y*. The following statement illustrates how these opposite operations work.

```
SELECT DECIMAL(ASIN(.997495),15,5), DECIMAL(SIN(1.5),15,5)
       FROM MYFILE
```

When executed, this statement returns values of 1.5 and .99749, respectively.

Note that we are making use of another function here. The DECIMAL function converts the supplied value to a decimal value. The supplied value can be any numeric format or a character string that evaluates to a number. The second parameter identifies the total length of the returned numeric value. The third parameter defines the precision and the number of decimal places shown. If the second and third optional parameters are omitted, the value returned will have zero decimal places. The example below illustrates using the DECIMAL function to convert a character string representation of a number to a 15-digit numeric field with two decimal places.

```
SELECT DECIMAL('123.159999',15,2)
       FROM MYFILE
```

The value returned by this SELECT statement would be 123.15. You'll notice that the resulting value is not rounded up. Any trailing decimals are simply truncated.

A series of functions is available for calculating arc tangent and inverse functions for tangents. The ATAN function calculates the arc tangent of the provided value. Its opposite function, TAN, calculates the tangent for the provided value. The ATANH and TANH functions return the hyperbolic arc tangent and hyperbolic tangent values, respectively. While these functions accept a single parameter between -1 and 1, the ATAN2 function returns the arc tangent based on two provided parameters, which represent *x* and *y* coordinates.

```
SELECT DECIMAL(ATAN(.5),15,5), DECIMAL(ATANH(.5),15,5),
        DECIMAL(ATAN2(5,3),15,5)
        FROM MYFILE
```

This example evaluates each of these functions. The values returned would be .46364, .54390, and .54041, respectively. Replacing the first two functions in our example with their opposite functions, the resulting values would be .54630 for the TAN function and .46211 for TANH.

The BIGINT function converts a value to a large integer. This value can be up to 31 numeric positions. The INTEGER and INT functions behave identically (for some functions, multiple versions exist for compatibility purposes). These two functions also convert the supplied value to an integer value, with a length of up to 15 numeric positions. The parameter supplied to the function can be a numeric value that is not an integer or a character value that represents a numeric value. The following sample statement illustrates multiple uses of the BIGINT function.

```
SELECT BIGINT('1250.5575'),BIGINT('1075723489760235'),
       BIGINT(107505230695704321.122585)
       FROM MYFILE
```

When executed, this example will return 1250, 1075723489760235, and 107505230695704321, respectively. Note that our first example illustrates a character representation of a floating point number. The second example shows a character representation of a long integer value, and the third field is a floating point value.

SQL supports two functions for rounding a numeric value to an integer. The CEILING function rounds the supplied value to the next highest integer value. Similarly, the FLOOR function rounds to the first integer value that is less than or equal to the supplied value. The following statement shows a sample of these two functions.

```
SELECT CEILING(25.475), FLOOR(25.575)
       FROM MyFile
```

The first column will return a value of 26, while the second will return 25. These functions allow you to force rounding in one direction or another based on your need.

Many of the geometric functions we've examined return a value that is a representation of an angle in radians. To convert this value to a number of degrees, we can use the DEGREES function. This function accepts a single parameter, which contains the radians value. Below is a sample of using this statement with the COS function.

```
SELECT DECIMAL(DEGREES(COS(.75)),15,5)
       FROM MyFile
```

When this statement is executed, the value returned would be 41.92268 degrees.

The DOUBLE, DOUBLE_PRECISION, and FLOAT functions convert the provided value into a floating point numeric value. The supplied parameter can contain a numeric or character string value. Below is an example of this statement.

```
SELECT DOUBLE(ORDQTY)
       FROM ORDERS
```

Assuming that the field of ORDQTY contained a value of 65.490, the value returned by any of these three functions would be 6.5489999999999995E+001.

The EXP function raises the natural logarithm "e" (approximately 2.71828182846) to the power specified on the supplied parameter. The statement below illustrates the use of this function.

```
SELECT EXP(ORDQTY)
       FROM ORDERS
```

If the value of the field ORDQTY was 6, the value returned by this function would be 403.428793492735110.

The LN function is the opposite function to the EXP function; it returns the natural logarithm for the supplied value. The example below illustrates the use of this function.

```
SELECT LN(403.428793492735110)
       FROM MYFILE
```

When executed, this statement returns a value of 6.

Similarly the LOG10 function returns the common logarithm (base 10) of the supplied value. The example below returns a value of 3.

```
SELECT LOG10(1000)
       FROM MYFILE
```

The MOD function calculates a remainder when the first parameter is divided by the second. The sample statement below illustrates how this function is used.

```
SELECT MOD(20,3)
       FROM MYFILE
```

In this example, 20 divided by 3 evaluates to 6 with a remainder of 2. As a result, 2 is the value returned by the function.

The MULTIPLY_ALT function is used as an alternative to performing multiplication operations using the asterisk (*) operator. The two values provided are multiplied by one another.

```
SELECT MULTIPLY_ALT(12,5)
       FROM MYFILE
```

As one might expect, this example returns a value of 60.

The PI function evaluates the value of pi (3.141592653589793). The example below calculates the circumference of a circle whose diameter is 5.

```
SELECT PI()*5
       FROM MYFILE
```

The result returned when this statement is executed is 15.7.

String Functions

Character strings and values can be manipulated using the string functions supported in SQL.

The CHAR function allows us to convert other field types to a character string value. When using this function, the first parameter defines the value to be converted. The second parameter is defined differently, depending on what type of value is identified by the first parameter. When a DATE or TIME field is being converted, the second parameter identifies the format for the converted date. Table 2.3 contains a list of the possible values.

Table 2.3: The CHAR Function Date Formats

Date Type	String Functions	Format
ISO	Industry Standards Organization format	(yyyy-mm-dd)
USA	USA date format	(mm/dd/yyyy)
EUR	European standard format	(dd.mm.yyyy)
JIS	Japanese Industrial Standard	(yyyy-mm-dd)
LOCAL	Based on the time/date format defined on the server	

When the CHAR function is used on a character or graphic field, the second parameter identifies the length of the resulting string from 1 to 32766. When the

CHAR function is used on an integer field, the second parameter is not used. When it's used on other numeric fields, the value should be specified as a field containing the single character to be used in place of the decimal point when the field is converted. The following statement shows samples of each of these conversions.

```
SELECT CHAR(DATE('12/12/2004'),ISO), CHAR('ABCDEFGHIJK',6),
   CHAR(123.45, ',')
       FROM SYSIBM.SYSDUMMY1
```

This statement uses the DATE function, which we'll examine later, to arrive at a date value. When executed, the statement will return values of 2004-12-12, ABCDEF, and 123,45. The date value is converted from USA format to ISO format. The character string is truncated to 6 characters based on the second parameter. With our third column, the decimal point is replaced by a comma.

The CHAR_LENGTH, CHARACTER_LENGTH, and LENGTH functions determine the length of a character string. When the parameter specified is a field name, the length of the field itself is returned. If a string literal is specified, the full length of that string, including trailing blanks, is returned. Following is a sample of using this function.

```
SELECT CHAR_LENGTH('123456      '), CHAR_LENGTH('123456')
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, the first column returns a value of 12, while the second column returns a value of 6. The VARCHAR function is used in the same way that CHAR is used except that the value returned is a VARCHAR field.

To join two string expressions into one, we use the CONCAT function. The first parameter specified is joined with the second parameter specified in the same way that two strings can be joined, using the double bar (||) concatenation operator. Below is a sample of using this function.

```
SELECT CONCAT('ABC','123'), 'ABC' || '123'
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, both columns return a value of ABC123.

The DIFFERENCE function determines how similar two string values are, based on the SOUNDEX function. To examine the DIFFERENCE function, we should first examine the SOUNDEX function itself. This function returns a four-character value, which is used to analyze the sound of a word. The example below returns the SOUNDEX values for two similar words.

```
SELECT SOUNDEX('TREE'), SOUNDEX('TRACE')
       FROM SYSIBM.SYSDUMMY1
```

Upon execution of this statement, the first column returns a value of T600, and the second returns a value of T620. If we change the string supplied to the first SOUNDEX function to 'TREES', the two values match. The DIFFERENCE function uses this logic to compare the two strings provided to the function. The value returned is a numeric value from 0 to 4, where 4 is the closest to a match and 0 is the furthest from a match. The statement below shows examples of each of the values.

```
SELECT DIFFERENCE('TREES','TRACE'),
       DIFFERENCE('TREE','TRACE'),
       DIFFERENCE('TIES','TRACE'),
       DIFFERENCE('APPLE','TRACE')
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, the values 4, 3, 2, and 1 will be returned. These two functions can be very useful, for example, when searching for a customer name where you are unsure of the spelling. The statement below gives an example of this.

```
SELECT CUSLNM, CUSFNM, CUSADD, CUSCTY, CUSSTE, CUSPHN
       FROM CUSNAMES
       WHERE DIFFERENCE(CUSLNM, 'JONSON')=4
```

Using this example, records will be returned for the names JOHNSON, JOHNSEN, and JENSEN, but not for JONES. You can use a lower numeric value to make the search less sensitive.

The DIGITS function is similar to the CHAR function. This function converts a numeric value to a character string value. The value returned to the string is unsigned, meaning that it is based on the absolute value of the numeric value supplied. The decimal point is also excluded from the string value. The statement below illustrates the use of this function.

```
SELECT DIGITS(-10123.858)
       FROM SYSIBM.SYSDUMMY1
```

When executed, this statement returns a value of 10123858. This function can be useful when you need to substring portions of a numeric field. For example, the following statement will take a date stored in an 8-digit numeric field as 20041231 and convert it to a displayable string representation of the date in mm/dd/yyyy format.

```
SELECT SUBSTR(DIGITS(DTEFLD),5,2) || '/' ||
       SUBSTR(DIGITS(DTEFLD),7,2) || '/' ||
       SUBSTR(DIGITS(DTEFLD),1,4)
       FROM MYFILE
```

Assuming that the value of DTEFLD is 20041231, when this statement is executed, a value of '12/31/2004' is returned.

The INSERT function inserts a specified string into a source string, starting at a specified position, while deleting the number of characters specified as length. The first parameter used on this function defines the source string. The second parameter defines the starting position at which the insertion is to occur within that source string. The third parameter defines the number of characters from the start position to delete from the source string prior to insertion. The final parameter identifies the string to be inserted. Below is an example of the syntax for this function.

```
SELECT INSERT('ABCDEF' ,3, 4, 'OUT')
       SYSIBM.SYSDUMMY1
```

When this statement is executed, the value 'ABOUT' is returned. If the string to be inserted is defined as null, the characters defined by the start position and length will be removed from the string altogether.

The GRAPHIC and VARGRAPHIC functions convert from character or numeric data to a value compatible with double-byte character data as is used for the Chinese or Japanese language. The result of either of these functions will be a field that is either a GRAPHIC or VARGRAPHIC data type, respectively. The statement below illustrates the use of both of these functions.

```
SELECT GRAPHIC('HELLO'), VARGRAPHIC('HELLO')
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, both columns return the value 'âHâEâLâLâO'.

The LCASE and LOWER functions convert the provided string to a lowercase representation of the same string. Below is an example of this function.

```
SELECT LCASE('ABC123')
       FROM SYSIBM.SYSDUMMY1
```

When executed, this statement returns a value of 'abc123'.

Similarly, the functions UCASE and UPPER convert a string to uppercase. Below is a sample of the UPPER function.

```
SELECT UPPER('Mike Faust')
      FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, the value 'MIKE FAUST' is returned.

The LEFT function extracts a specified number of characters from the left side of the provided string value. The first parameter identifies the source string, while the second defines the number of characters to be extracted. Below is a sample statement using this function.

```
SELECT LEFT('ABC123', 3)
       FROM SYSIBM.SYSDUMMY1
```

This statement will return the value 'ABC' when executed.

The RIGHT function is similar to this function with the exception that it extracts from the right side of the defined string. Below is a modified version of the previous statement using RIGHT.

```
SELECT RIGHT('ABC123', 3)
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, the value '123' is returned.

The SUBSTR (or SUBSTRING) function is also used to extract characters from a source string. This function, however, accepts three parameters: The first is the source string, the second defines the starting position for the characters to be extracted, and the third defines the number of characters to be extracted. The statement below shows an example of this function.

```
SELECT SUBSTR('ABC123', 3, 2)
       FROM SYSIBM.SYSDUMMY1
```

When executed, this statement returns a value of 'C1'.

The LOCATE function searches for the string defined on its first parameter within the source string defined on its second. An optional third parameter can be specified to identify the start point within the source string to search. Below is an example of this function's use.

```
SELECT LOCATE('AB', 'ABCABDABE'), LOCATE('AB', 'ABCABDABE', 3)
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, the first column returns a value of 1. The second column returns a value of 2, which is the location where the string is found, taking the start position into consideration. To determine the actual start position in this case, we need to take the start position value (3), add the value returned (2), and subtract 1, giving us a value of 4.

The POSSTR and POSITION functions perform a similar function to LOCATE. However, these functions accept only the search string and source string values. A start position cannot be specified with these functions. Below is an example.

```
SELECT POSITION('AB' IN 'BCABDABE'), POSSTR('BCABDABE', 'AB')
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, both of these functions return a value of 3.

We can remove any leading blanks from a string value using the LTRIM function. This function removes all blank spaces from the left side of the supplied string value, effectively left-adjusting that string. Below is a sample of this function.

```
SELECT LENGTH(LTRIM('   ABC'))
       FROM SYSIBM.SYSDUMMY1
```

To illustrate that the resulting string has been changed, I've combined the LENGTH function with the LTRIM function. When executed, this statement returns a value of 3.

Similarly, the RTRIM function removes all trailing blanks from the specified string. The example below illustrates this function's use.

```
SELECT LENGTH(RTRIM('ABC 123     '))
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, the trailing blanks will be removed, and a value of 7 is returned. Note that the embedded blank character is not affected.

The TRIM and STRIP functions also remove characters from a specified string, but both have more functionality than the other two functions. When these functions are used with a source string only, the value returned is stripped of both leading and trailing blanks. An example of this is shown below.

```
SELECT LENGTH(TRIM('   1234   '))
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, the value 4 is returned. Optional modifiers allow us to use the TRIM function to remove leading and/or trailing blanks or

other characters from the supplied string. The example below can be used to remove leading zeros from the defined string.

```
SELECT TRIM(LEADING '0' FROM '000123400'),
       TRIM(TRAILING '0' FROM '000123400'),
       TRIM(BOTH '0' FROM '000123400')
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, values returned are '123400', '0001234', and '1234', respectively.

The REPEAT function creates a string containing the expression supplied on the first parameter repeated the number of times defined on the second. The statement below illustrates this statement's use.

```
SELECT REPEAT('A1B2C3', 3)
       FROM SYSIBM.SYSDUMMY1
```

When executed, this statement returns a value of 'A1B2C3A1B2C3A1B2C3'.

Similarly, the SPACE function returns a number of blank spaces as specified on the required parameter. The statement below illustrates the use of this function.

```
SELECT SPACE(32)
       FROM SYSIBM.SYSDUMMY1
```

This statement returns a string value containing 32 blank spaces.

The REPLACE function allows us to replace a search string specified on the first parameter within a source string specified on the second parameter with the replacement string specified on the third. The statement below shows four examples of different uses for this statement.

```
SELECT REPLACE('XY', 'XYZ', 'BI'), REPLACE('XY', 'XYZ', ''),
       REPLACE('XY', 'XYZ', 'JAZ'), REPLACE('XY', 'ABC', 'DE'),
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, the first column will replace ‘XY’ in ‘XYZ’ with ‘BI’, resulting in ‘BIZ’. The second column will replace ‘XY’ in ‘XYZ’ with a zero length string, resulting in ‘Z’. The third column replaces ‘XY’ in ‘XYZ’ with ‘JAZ’, resulting in ‘JAZZ’. Finally, the fourth column doesn’t locate ‘XY’ in ‘ABC’ and as a result returns the original value of ‘ABC’.

Date/Time Functions

SQL contains a set of built-in functions that allow us to manipulate fields containing date and/or time values.

The functions CURDATE and CURTIME allow us to retrieve the current date and current time, respectively. Both functions have no parameters. The statement below shows a sample of using these functions within an INSERT statement.

```
INSERT INTO TRANS(ITEM, QTY, TRNDTE, TRNTIM)
       VALUES('ABC123', 5000, CURDATE(), CURTIME())
```

This statement adds a record to the table named TRANS, which contains four fields. The last two fields represent the date and time of the transaction being added. The DATE function converts a string representation of a date to a date value. We used this function earlier when examining the CHAR function. The date supplied to the DATE function must be a value date in the format as defined for the job. If the job date format is ISO, then a date in the format yyyy-mm-dd must be supplied. The statement below uses this function combined with the CURDATE function.

```
SELECT *
       FROM MYFILE
       WHERE CURDATE>DATE(STRDTE)
```

Assuming that the field MYFILE has a field called STRDTE stored in the date format defined for the current job, this statement will select all records for dates earlier than the current date.

Similarly, the TIME function converts a time stored in a string to a time value. The statement below illustrates the use of this function.

```
SELECT TIME('12.00.00') AS Noon
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, the value returned is '12:00:00'. Note that the time separator has been changed from the decimal point to the colon character. This is based on the value defined for the time separator on the job.

A group of functions allows us to extract a given part of a date or time value. The DAY, WEEK, MONTH, QUARTER, and YEAR functions extract the portion of a date field suggested by their name. The example below illustrates this by selecting each component of the CURDATE value.

```
SELECT DAY(CURDATE()), WEEK(CURDATE()), MONTH(CURDATE()),
       QUARTER(CURDATE()), YEAR(CURDATE())
       FROM SYSIBM.SYSDUMMY1
```

When executed, this statement returns columns containing the current date (1 thru 31), week number (1 thru 54), month number (1 thru 12), quarter number (1 thru 4), and year number ().

Similarly, a set of time functions allows us to extract pieces of a time value. WEEK defines the week as starting on Sunday. January 1 always falls in week 1. As an alternative to WEEK, SQL also supports a WEEK_ISO function. This function returns a value from 1 to 53. For WEEK_ISO, the week begins on Monday and is defined as the first week containing a Thursday. This means that if January 1 falls on a Friday, Saturday, or Sunday, then week 1 will be the following week.

The DAYOFWEEK and DAYOFWEEK_ISO functions return the day of the week from 1 to 7. The difference between these two functions is that the DAYOFWEEK value of 1 indicates Sunday, and the DAYOFWEEK_ISO value of 1 indicates Monday.

Similarly, the DAYOFMONTH function returns the day within the month (1 to 31) for the supplied date value. This function is similar to the DAY function.

The DAYOFYEAR function returns a value from 1 to 366 that identifies the day of the year for the supplied date.

The DAYNAME function returns a string containing the name of the day of the week for the supplied date value. The example below illustrates these functions.

```
SELECT DAYOFMONTH(CURDATE()), DAYOFYEAR(CURDATE()), DAYNAME
   (CURDATE()),
        FROM SYSIBM.SYSDUMMY1
```

If this statement is executed on April 15, 2005, the values returned will be '15', '105', and 'Friday'.

SQL also supports functions to allow us to extract pieces of a time field. The functions HOUR, MINUTE, SECOND, and MICROSECOND each extract the portion of the specified time value indicated by their names. Assuming that the value provided to the HOUR function is a time or timestamp or a character representation of a time, the value returned will be between 0 and 24. Each of these functions accepts a time value, a timestamp value, or a string representation of a time. The statement below extracts the hour, minute, and second of the time 12:15:45.

```
SELECT HOUR('12:15:45'), MINUTE('12:15:45'), SECOND('12:15:45')
       FROM SYSIBM.SYSDUMMY1
```

When executed, this statement returns values of '12', '15', and '45', respectively.

The EXTRACT function performs a similar function to each of the individual time and date extraction functions. The first parameter specifies the portion of the date/time value to be extracted. Values of YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND are valid for this parameter. The second parameter supplies the date, time, or timestamp value from which the data is to be extracted. The sample statement below extracts the hour from the current time.

```
SELECT EXTRACT(HOUR, '12:15:45')
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, a value of 12 is returned.

The JULIAN_DAY function returns a value representing the number of days since the start of the Julian calendar (1/1/4713 B.C.). The statement below illustrates the use of this function.

```
SELECT JULIAN_DAY('12/31/2005')
       FROM SYSIBM.SYSDUMMY1
```

When executed, this statement returns the value 2453736.

The MIDNIGHT_SECONDS function is similar to JULIAN_DAY. This function returns the number of seconds between midnight and the time provided to the function's parameter. The statement below illustrates the use of this function.

```
SELECT MIDNIGHT_SECONDS('12:30:00')
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, a value of 45000 seconds is returned.

The TIMESTAMP function converts string representations of a timestamp or date and time values to a timestamp value. The statement below illustrates two different uses of this statement.

```
SELECT TIMESTAMP(DATE('12/31/2005'),TIME('12:15:00')),
       TIMESTAMP('2005-12-31-12.15.00.000000')
       FROM SYSIBM.SYSDUMMY1
```

When executed, this statement returns timestamp values of 2005-12-31-12.15.00.000000 for both columns. Note that the string representation of the timestamp is specified in the format yyyy-mm-dd-hh.mn.ss.milsec.

The TIMESTAMPDIFF function allows us to express the difference between two timestamps. The first parameter is an integer value used to identify the interval in which the difference between the timestamps should be given. Table 2.4 gives a list of the possible values for this parameter.

Table 2.4: TIMESTAMPDIFF Values

Value	Time Interval	Value	Time Interval
1	Microseconds	32	Weeks
2	Seconds	64	Months
4	Minutes	128	Quarters
8	Hours	256	Years
16	Days		

The second parameter is a 22-character string containing the result of a subtraction operation performed on two timestamp values. The SQL statement below shows how this is accomplished.

```
SELECT TIMESTAMPDIFF(8, CAST(NOW() - TIMESTAMP(
       '2005-01-01-00.00.00.000000') AS CHAR(22)))
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, it returns the number of hours between 1/1/2005 at midnight and the current time.

Similarly, the DAYS function converts a given date or timestamp value or a string representation of either of those into a number of days. Generally, this function is used to determine duration in days. The example below illustrates using this function to perform duration-based calculations.

```
SELECT DAYS(ENDATE)-DAYS(STDATE), DATE(DAYS(NOW()) + 5)
       FROM SYSIBM.SYSDUMMY1
```

Assuming that the field ENDATE is 12/31/2005 and STDATE is 6/15/2005, the value returned for this column will be 199. The second example uses the DATE function to convert the DAYS value (plus our increment of 5 additional days) back to a date format. When the statement is executed, this column will result in the current date plus five days.

Datalink Functions

Datalink fields and values allow us to define links from within our database to files outside of our database. These links are defined using URL addressing. They can be files stored on an FTP server (http://ftp.ibm.com/anyfile.txt), on a Web server (http://www.geocities.com/mikeffaust/index.html), or in a local or network share accessible file (file://c:\myfile.txt). The datalink field contains the datalink value itself in addition to an optional comment.

The DLVALUE function creates a datalink value from the string defined on the function's first parameter. The optional second parameter defines the type of link being created. Currently, the only value supported is the value 'URL'. The optional third parameter defines the comment associated with the datalink. The statement below is a sample of using the DLVALUE function.

```
INSERT INTO MYLIB.WEBPAGES(WEBSTE)
   VALUES(DLVALUE('http://www.geocities.com/mikeffaust/ index.html',
   'URL', 'Mike Faust's Tips Website'))
```

When this statement is executed, a datalink value is created for the Web page shown. The datalink comment is set to "Mike Faust's Tips Website."

In addition to the DLVALUE function, several other functions can be used to read information about a given datalink value. The DLCOMMENT function reads the comment from an associated datalink value. This function's only parameter is the name of the datalink field or a datalink value. The statement below shows an example of using this function in a SELECT statement.

```
SELECT DLCOMMENT(WEBSTE)
       FROM MYLIB.WEBPAGES
```

When executed, this statement returns a list containing the comment data for the datalink field WEBSTE in each record of our table. Using our earlier example, this statement would return "Mike Faust's Tips Website."

The DLLINKTYPE function returns the link type for the datalink value provided on the function's first parameter. The example below illustrates how this function is used.

```
SELECT DLLINKTYPE(WEBSTE)
       FROM MYLIB.WEBPAGES
```

Since 'URL' is the only value link type currently supported on DB2, this function should always return the value 'URL'.

The DLURLCOMPLETE function returns the full URL from the supplied datalink value. The statement below illustrates the use of this function.

```
SELECT DLURLCOMPLETE(WEBSTE)
       FROM MYLIB.WEBPAGES
```

Using the data from our previous example, when this statement is executed, the URL 'HTTP://WWW.GEOCITIES.COM/DEFAULT/MIKEFFAUST/INDEX.HTML' is returned. The datalink value is always returned as an uppercase representation of the value supplied.

The DLURLPATH function returns the path and file portion of a datalink value, meaning that it strips out the server name portion of the URL. The sample statement below illustrates the use of this function.

```
SELECT DLURLPATH(WEBSTE)
       FROM MYLIB.WEBPAGES
```

Again using our earlier sample data, when this statement is run, the value returned will be '/DEFAULT/MIKEFFAUST/INDEX.HTML'. The result of this function can include a file access token as a series of asterisks where appropriate.

The DLURLPATHONLY function also returns the path and file portion of a datalink URL but does not include file access tokens.

The DLURLSCHEME function returns the scheme for the datalink provided. This value identifies the type of URL being provided. The sample statement below illustrates the use of this function.

```
SELECT DLURLSCHEME(DLVALUE('https://192.168.50.11/myfile.txt'))
       FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, the value 'HTTPS' is returned.

The DLURLSERVER function returns the portion of the URL associated with the datalink value provided that identifies the server containing the linked document. The sample below shows how this function is used.

```
SELECT
DLURLSERVER(DLVALUE('http://www.geocities.com/mikeffaust/index.html'))
     FROM SYSIBM.SYSDUMMY1
```

When executed, this statement returns the value 'WWW.GEOCITIES.COM'.

Combined, these functions give you full control of a datalink field.

Logic Functions

The logical functions in DB2 SQL allow us to perform AND and OR operations on supplied values. The LAND function accepts two or more string values and performs a logical AND on those two strings. Below is a sample of this statement's use.

```
SELECT HEX('HELLO!'), HEX('FELLOW'), HEX(LAND('HELLO!', 'FELLOW'))
       FROM SYSIBM.SYSDUMMY1
```

This statement takes the string HELLO! and does a logical AND with FELLOW. To get a better idea of the actual function being performed, I've shown each of

the values in hex before the AND, while also converting the result of the AND function to hex. The result returned for the three columns is x‘C8C5D3D3D65A’, x‘C6C5D3D3D6E6’, and x‘C0C5D3D3D642’, respectively.

The LNOT function returns the logical NOT value for the provided string expression. The statement below uses this statement to evaluate the logical NOT for the hex value ‘FF’.

```
SELECT HEX(LNOT(x‘FF’))
       FROM SYSIBM.SYSDUMMY1
```

Note that again I use the HEX function to convert the value returned to hex for viewing. When executed, this statement returns the value ‘00’.

The LOR function returns the logical OR value for the supplied string expressions. The sample below calculates the logical OR for hex values ‘3C’ and ‘C3’.

```
SELECT HEX(LOR(x‘3C’, x‘C3’))
       FROM SYSIBM.SYSDUMMY1
```

Once again, I convert the resulting value to a hex value for easy viewing. When this statement is executed, the result returned will be the hex value ‘FF’.

The XOR function returns the logical XOR value for the provided string values. The example below illustrates this function’s use.

```
SELECT HEX(XOR(x‘FC’, x‘3F’))
       FROM SYSIBM.SYSDUMMY1
```

As you can see, we are again using hex values in this example, this time calculating the XOR value for hex ‘FC’ and hex ‘3F’. The resulting value returned is the hex value ‘C3’.

Miscellaneous Functions

The remaining functions fall into what I’ll call “other” functions. The BLOB, CLOB, and DBCLOB functions are used to store large object (LOB) data types,

such as image files, sound files, videos, or other documents, each of which can have up to 2 gigabytes of data. The BLOB function is used to access a binary LOB. CLOB is used with single-byte character LOBs, and DBCLOB is used with LOBs stored as double-byte character. Each of these functions converts a character string into a LOB.

The first parameter on the function contains the string data. The second parameter, which is optional, provides the length of the LOB to be created.

These functions are usually used to append data to an existing LOB value. The following sample statement illustrates this using the BLOB function.

```
SELECT BLOB('This is LOB Data' CONCAT MYBLOB)
       FROM MYFILE
```

When this statement is executed, the column defined using our function will contain the text shown in addition to the object stored in the field MYBLOB. The value returned by this function will show '*POINTER', indicating that the value is a pointer to the location of the object itself. The same syntax is true for the CLOB and DBCLOB functions.

The COALESCE and IFNULL functions return the first non-null value in the list of supplied parameters. The example below illustrates the use of this function to return a zero value if the value of the field COSTAM is null.

```
SELECT IFNULL(COSTFILE.COSTAM,0) AS COST
       FROM MYFILE LEFT JOIN COSTFILE ON MYFILE.ITEM =
           COSTFILE.ITEM
```

Note that in this example I am using a left join expression to indicate that all records from the file on the left side of the join should be included and any matching records from the table on the right side of the expression should be included. This means that we can have records where no record is returned for the table COSTFILE. In that situation, the value of the field COSTFILE.COSTAM will be null. This function prevents the statement from returning a null value for this column.

The NULLIF function performs the opposite function to IFNULL. NULLIF returns a null value if the two values provided on the function's parameters are equal. The statement below makes use of this function.

```
SELECT NULLIF(COST1,0)
       FROM MYTABLE
```

When this example is executed, the function will return null for any case where COST1 is 0; otherwise, the function will return the value of COST1.

The VALUE function returns the first value in the list of values supplied that does not evaluate to NULL. Ultimately, this is similar to IFNULL with additional functionality. The statement below illustrates this function's use.

```
SELECT VALUE(PRICEFILE.PRICE, COSTFILE.COSTAM, 0) AS DOLLARS
       FROM MYFILE LEFT JOIN COSTFILE ON MYFILE.ITEM =
  COSTFILE.ITEM LEFT JOIN PRICEFILE ON MYFILE.ITEM =
  PRICEFILE.ITEM
```

When this statement is executed, the first non-null value is returned. If either of the tables COSTFILE or PRICEFILE does not contain a matching item record, the value of their fields will evaluate to null. If both fields evaluate to null, a value of 0 is returned.

The DATABASE function returns the name of the current SQL database. This value will generally be the system name of your iSeries or i5. This function has no parameters. The statement below illustrates its use.

```
SELECT DATABASE()
        FROM SYSIBM.SYSDUMMY1
```

When this statement is executed, a value representing the name of the current database is returned.

The DATAPARTITIONNAME function returns the data partition name for each row returned by a statement. Partitioning allows us to store data in multiple members yet treat those members as a single table from within SQL. The

DATAPARTITIONNAME value returned is the name of the partition or member containing the current row of data. Similarly, the DATAPARTITIONNUM function determines the partition number containing the current row. The statement below returns the partition name and number for each row in the table supplied on the first parameter.

```
SELECT DATAPARTITIONNAME(S1), DATAPARTITIONNUM(S1)
       FROM SYSIBM.SYSDUMMY1 S1
```

While the DBPARTITIONNAME and DBPARTITIONNUM functions look similar to the two I've just explained, they are not the same functions. These two functions give the partition names related to data spread across multiple systems. The DBPARTITIONNAME function returns the relational database name containing the current row, and the NODENAME function performs the exact same task. The DBPARTITIONNUM function returns the database partition number for the current row. If the table defined on the function's parameter is not a distributed table, the function returns a value of 0. The following statement illustrates the use of these two functions.

```
SELECT DBPARTITIONNAME(S1), DBPARTITIONNUM(S1)
       FROM SYSIBM.SYSDUMMY1 S1
```

DB2 SQL supports a set of functions for encrypting and decrypting data. The ENCRYPT_RC2 function uses the RC2 encryption algorithm to encrypt the data supplied on its first parameter. This function supports two additional parameters that define a password, which is required to decrypt the data, and a hint value to assist in retrieving a lost password. A password can also be specified using the SET ENCRYPTION PASSWORD statement. The following statement is an example of using the ENCRYPT_RC2 function to insert encrypted account number values into a table.

```
INSERT INTO ACCOUNTS(ACCTNM)
       VALUES(ENCRYPT_RC2('123456789', 'EAGLE', 'TALON')
```

When executed, this statement will insert a row containing the encrypted value 123456789 with a password of EAGLE and a hint value of TALON. To decrypt a value encrypted in this method, we can choose from the following functions.

1. DECRYPT_BIT decrypts bit data.
2. DECRYPT_BINARY decrypts binary data.
3. DECRYPT_CHAR decrypts single-byte character data.
4. DECRYPT_DB decrypts double-byte character data.

The first parameter for each of these functions represents the field containing the encrypted data. The second parameter contains the encryption password or the special value DEFAULT, which identifies that the password defined using SET ENCRYPTION PASSWORD statement should be used. The optional third parameter can be used with single- and double-byte character fields to define the code page. The statement below illustrates using the DECRYPT_CHAR function to decrypt the value inserted in the previous example.

```
SELECT DECRYPT_CHAR(ACCTNM, 'EAGLE')
       FROM ACCOUNTS
```

When this statement is executed, the value '123456789' is returned.

The GETHINT function retrieves the hint text for the provided encryption field. The statement below illustrates using this function with the sample data we created earlier.

```
SELECT GETHINT(ACCTNM)
       FROM ACCOUNTS
```

When executed, this statement returns the value 'TALON'.

The HASH function returns a value indicating the partition number containing the values specified. The value returned is an integer between 0 and 1023. Below is a sample of this statement.

```
SELECT HASH(IBMREQD)
      FROM SYSIBM.SYSDUMMY1
```

Similarly, the HASHED_VALUE function returns the partition number. However, this function returns the value based on the current row in the table specified on

the function's only parameter. Again, the value returned is an integer number between 0 and 1023. Below is a sample of this statement. Note the S1 identifier after the table name, which is used as an alias for this table in the HASH function.

```
SELECT HASH(S1)
       FROM SYSIBM.SYSDUMMY1 S1
```

We used the HEX function in earlier examples. This function returns a hexadecimal representation of a string value. The statement below illustrates using this function to convert three different values to hex.

```
SELECT HEX(12), HEX('HELLO'), HEX(DATE('12/15/2005'))
       FROM SYSIBM.SYSDUMMY1
```

This example converts a numeric value, a string value, and a date value to hexadecimal. When executed, this statement returns x'0000000C', x'C8C5D3D3D6', and x'002570D8', respectively, for the columns above.

The function IDENTITY_VAL_LOCAL determines the most recent value of an identity column. The statement below assumes that the table MYTABLE contains an identity column defined using the modifiers "INTEGER GENERATED ALWAYS AS IDENTITY" when creating the column on the CREATE TABLE statement.

```
SELECT IDENTITY_VAL_LOCAL()
       FROM MYTABLE
```

Assuming that the last record added to MYTABLE had an identity column value of 15, this function would return 15.

The RRN function is somewhat similar to the IDENTITY_VAL_LOCAL function. It returns the relative record number for the current record in the table defined on the function's single parameter. The example below illustrates the use of the RRN function.

```
SELECT RRN(S1)
       FROM SYSIBM.SYSDUMMY1 S1
```

When executed on the sample table SYSIBM.SYSDUMMY1, this function returns a single record showing a value of 1. If this statement is executed on a larger table, a value is returned for each record in the table.

The MAX and MIN scalar functions perform similar tasks to their columnar cousins. These two functions return the highest value in the list of values and/or fields provided on the function's parameters. The statement below illustrates the use of both functions.

```
SELECT MIN(COST1, COST2, COST3), MAX(COST1, COST2, COST3)
       FROM MYTABLE
```

When this statement is executed, the first column will return a value representing the lowest of the fields COST1, COST2, and COST3. The second will return the greatest of the fields COST1, COST2, and COST3.

Functions vs. Procedures

Now that you have a good understanding of built-in functions and what they can do, it's time to move on to chapter 3 to learn more about stored procedures.