TIBCO® Data Migrator

Functions Reference

Release 8207
March 2021
DN3502239.0321

Contents

Chapter 1

Functions Overview

Functions provide a convenient way to perform certain calculations and manipulations. They operate on one or more arguments and return a single value that is assigned to an *output_format*. The returned value can be stored in a field, assigned to a Dialogue Manager variable, used in an expression or other processing, or used in a selection or validation test. These functions can be used in source and target objects.

In this chapter:

Function Arguments

All function arguments except the last one are *input arguments*. The formats for these arguments are described with each function. Unless specified, every input argument can be provided as one of the following:

- ❑ A literal (that is, a number for numeric formats or a character string enclosed in single quotation marks for alphanumeric formats).
- ❑ A field of the correct format.
- ❑ A variable assigned by a Dialogue Manager command.
- ❑ An expression result evaluated in the correct format.

The *output* argument is the last function argument. With few exceptions, it is a required argument whose only goal is to provide *a format* for the output of a function. It is *not* a field to put the result in. The format can be provided as either:

- ❑ A character string enclosed in single quotation marks.
- ❑ A field name whose format is to be used.

This field is the one to which the result of the expression evaluation is assigned. If the output_format is alphanumeric, its size should be large enough to fit the function output and avoid truncation; excessive size causes the output to be padded with blanks.

Note: With CDN ON, numeric function arguments must be delimited by a comma followed by a space.

Function Categories

Functions are grouped into the following areas:

- *Character Functions*
- *Variable Length Character Functions*
- *Character Functions for DBCS Code Pages*
- *Data Source and Decoding Functions*
- *Date Functions*
    - *Using Standard Date Functions*
    - *Using Legacy Date Functions*
- *Date-Time Functions*
- *Format Conversion Functions*
- *Numeric Functions*
- *System Functions*

Character Chart for ASCII and EBCDIC

This chart shows the primary printable characters in the ASCII and EBCDIC character sets and their decimal equivalents. Extended ASCII codes (above 127) are not included.

Decimal	ASCII		EBCDIC	
33	!	exclamation point		
34	"	quotation mark		
35	#	number sign		
36	$	dollar sign		
37	%	percent		

Decimal	ASCII		EBCDIC	
38	&	ampersand		
39	'	apostrophe		
40	(	left parenthesis		
41	)	right parenthesis		
42	*	asterisk		
43	+	plus sign		
44	,	comma		
45	-	hyphen		
46	.	period		
47	/	slash		
48	0	0		
49	1	1		
50	2	2		
51	3	3		
52	4	4		
53	5	5		
54	6	6		
55	7	7		
56	8	8		
57	9	9		
58	:	colon		
59	;	semicolon		

Decimal	ASCII		EBCDIC	
60	<	less-than sign		
61	=	equal sign		
62	>	greater-than sign		
63	?	question mark		
64	@	at sign		
65	A	A		
66	B	B		
67	C	C		
68	D	D		
69	E	E		
70	F	F		
71	G	G		
72	H	H		
73	I	I		
74	J	J	¢	cent sign
75	K	K	.	period
76	L	L	<	less-than sign
77	M	M	(	left parenthesis
78	N	N	+	plus sign
79	O	O	|	logical or
80	P	P	&	ampersand
81	Q	Q		

Decimal	ASCII		EBCDIC	
82	R	R		
83	S	S		
84	T	T		
85	U	U		
86	V	V		
87	W	W		
88	X	X		
89	Y	Y		
90	Z	Z	!	exclamation point
91	[	opening bracket	$	dollar sign
92	\	back slant	*	asterisk
93	]	closing bracket	)	right parenthesis
94	^	caret	;	semicolon
95	_	underscore	¬	logical not
96	`	grave accent	-	hyphen
97	a	a	/	slash
98	b	b		
99	c	c		
100	d	d		
101	e	e		
102	f	f		
103	g	g		

Decimal	ASCII		EBCDIC	
104	h	h		
105	i	i		
106	j	j		
107	k	k	,	comma
108	l	l	%	percent
109	m	m	_	underscore
110	n	n	>	greater-than sign
111	o	o	?	question mark
112	p	p		
113	q	q		
114	r	r		
115	s	s		
116	t	t		
117	u	u		
118	v	v		
119	w	w		
120	x	x		
121	y	y		
122	z	z	:	colon
123	{	opening brace	#	number sign
124	|	vertical line	@	at sign
125	}	closing brace	'	apostrophe

Decimal	ASCII		EBCDIC	
126	~	tilde	=	equal sign
127			"	quotation mark
129			a	a
130			b	b
131			c	c
132			d	d
133			e	e
134			f	f
135			g	g
136			h	h
137			i	i
145			j	j
146			k	k
147			l	l
148			m	m
149			n	n
150			o	o
151			p	p
152			q	q
153			r	r
162			s	s
163			t	t

Decimal	ASCII		EBCDIC	
164			u	u
165			v	v
166			w	w
167			x	x
168			y	y
169			z	z
185			`	grave accent
193			A	A
194			B	B
195			C	C
196			D	D
197			E	E
198			F	F
199			G	G
200			H	H
201			I	I
209			J	J
210			K	K
211			L	L
212			M	M
213			N	N
214			O	O

Decimal	ASCII		EBCDIC	
215			P	P
216			Q	Q
217			R	R
226			S	S
227			T	T
228			U	U
229			V	V
230			W	W
231			X	X
232			Y	Y
233			Z	Z
240			0	0
241			1	1
242			2	2
243			3	3
244			4	4
245			5	5
246			6	6
247			7	7
248			8	8
249			9	9

Chapter 2

Simplified Analytic Functions

The analytic functions enable you do perform calculations and retrievals using multiple rows in the internal matrix.

In this chapter:

FORECAST_MOVAVE: Using a Simple Moving Average

A simple moving average is a series of arithmetic means calculated with a specified number of values from a field. Each new mean in the series is calculated by dropping the first value used in the prior calculation, and adding the next data value to the calculation.

Simple moving averages are sometimes used to analyze trends in stock prices over time. In this scenario, the average is calculated using a specified number of periods of stock prices. A disadvantage to this indicator is that because it drops the oldest values from the calculation as it moves on, it loses its memory over time. Also, mean values are distorted by extreme highs and lows, since this method gives equal weight to each point.

Predicted values beyond the range of the data values are calculated using a moving average that treats the calculated trend values as new data points.

The first complete moving average occurs at the nth data point because the calculation requires *n* values. This is called the lag. The moving average values for the lag rows are calculated as follows: the first value in the moving average column is equal to the first data value, the second value in the moving average column is the average of the first two data values, and so on until the nth row, at which point there are enough values to calculate the moving average with the number of values specified.

Syntax: How to Calculate a Simple Moving Average Column

```
FORECAST_MOVAVE(display, infield, interval,
 npredict, npoint1)
```

where:

display

Keyword

Specifies which values to display for rows of output that represent existing data. Valid values are:

- **INPUT_FIELD.** This displays the original field values for rows that represent existing data.
- **MODEL_DATA.** This displays the calculated values for rows that represent existing data.

Note: You can show both types of output for any field by creating two independent COMPUTE commands in the same request, each with a different display option.

infield

Is any numeric field. It can be the same field as the result field, or a different field. It cannot be a date-time field or a numeric field with date display options.

interval

Is the increment to add to each sort field value (after the last data point) to create the next value. This must be a positive integer. To sort in descending order, use the BY HIGHEST phrase. The result of adding this number to the sort field values is converted to the same format as the sort field.

For date fields, the minimal component in the format determines how the number is interpreted. For example, if the format is YMD, MDY, or DMY, an interval value of 2 is interpreted as meaning two days. If the format is YM, the 2 is interpreted as meaning two months.

npredict

Is the number of predictions for FORECAST to calculate. It must be an integer greater than or equal to zero. Zero indicates that you do not want predictions, and is only supported with a non-recursive FORECAST.

npoint1

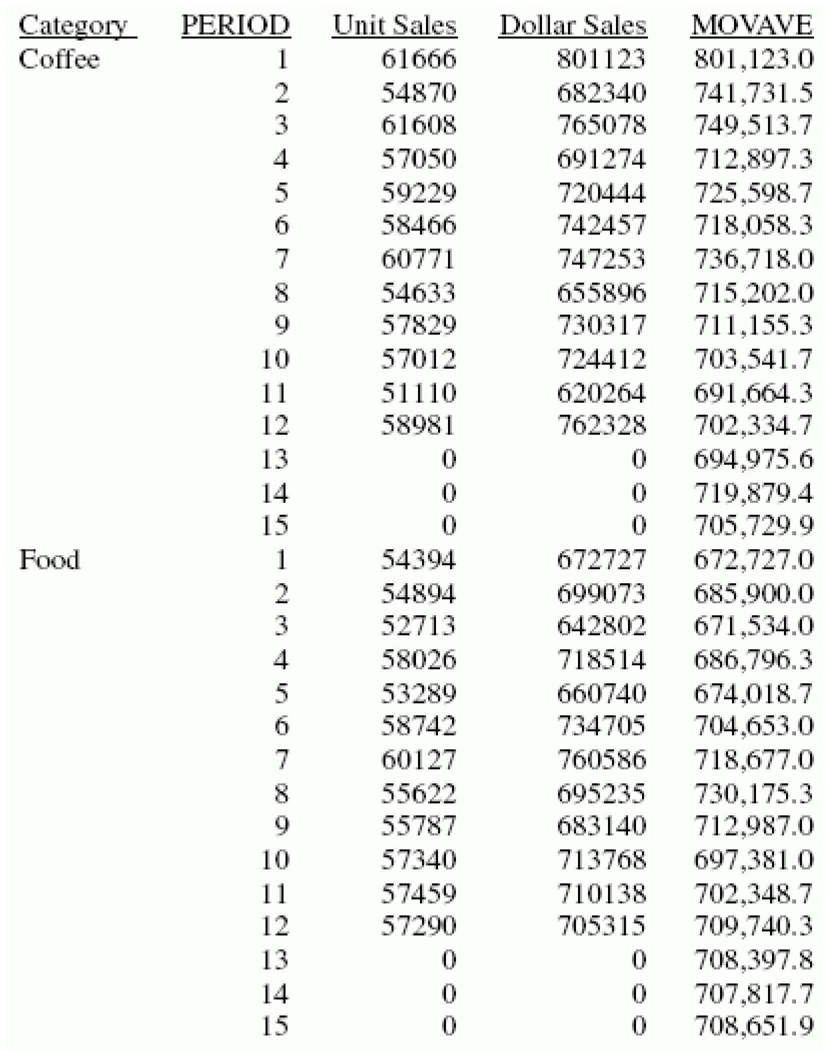
Is the number of values to average for the MOVAVE method.

Example: Calculating a New Simple Moving Average Column

This request defines an integer value named PERIOD to use as the independent variable for the moving average. It predicts three periods of values beyond the range of the retrieved data. The MOVAVE column on the report output shows the calculated moving average numbers for existing data points.

```
DEFINE FILE GGSALES
SDATE/YYM = DATE;
SYEAR/Y = SDATE;
SMONTH/M = SDATE;
PERIOD/I2 = SMONTH;
END
TABLE FILE GGSALES
SUM UNITS DOLLARS
COMPUTE  MOVAVE/D10.1= FORECAST_MOVAVE(MODEL_DATA, DOLLARS,1,3,3);
BY CATEGORY BY PERIOD
WHERE SYEAR EQ 97 AND CATEGORY NE 'Gifts'
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is:



Category	PERIOD	Unit Sales	Dollar Sales	MOVAVE
Coffee	1	61666	801123	801,123.0
	2	54870	682340	741,731.5
	3	61608	765078	749,513.7
	4	57050	691274	712,897.3
	5	59229	720444	725,598.7
	6	58466	742457	718,058.3
	7	60771	747253	736,718.0
	8	54633	655896	715,202.0
	9	57829	730317	711,155.3
	10	57012	724412	703,541.7
	11	51110	620264	691,664.3
	12	58981	762328	702,334.7
	13	0	0	694,975.6
	14	0	0	719,879.4
	15	0	0	705,729.9
Food	1	54394	672727	672,727.0
	2	54894	699073	685,900.0
	3	52713	642802	671,534.0
	4	58026	718514	686,796.3
	5	53289	660740	674,018.7
	6	58742	734705	704,653.0
	7	60127	760586	718,677.0
	8	55622	695235	730,175.3
	9	55787	683140	712,987.0
	10	57340	713768	697,381.0
	11	57459	710138	702,348.7
	12	57290	705315	709,740.3
	13	0	0	708,397.8
	14	0	0	707,817.7
	15	0	0	708,651.9

In the report, the number of values to use in the average is 3 and there are no UNITS or DOLLARS values for the generated PERIOD values.

Each average (MOVAVE value) is computed using DOLLARS values where they exist. The calculation of the moving average begins in the following way:

❑ The first MOVAVE value (801,123.0) is equal to the first DOLLARS value.

- The second MOVAVE value (741,731.5) is the mean of DOLLARS values one and two: (801,123 + 682,340) /2.
- The third MOVAVE value (749,513.7) is the mean of DOLLARS values one through three: (801,123 + 682,340 + 765,078) / 3.
- The fourth MOVAVE value (712,897.3) is the mean of DOLLARS values two through four: (682,340 + 765,078 + 691,274) /3.

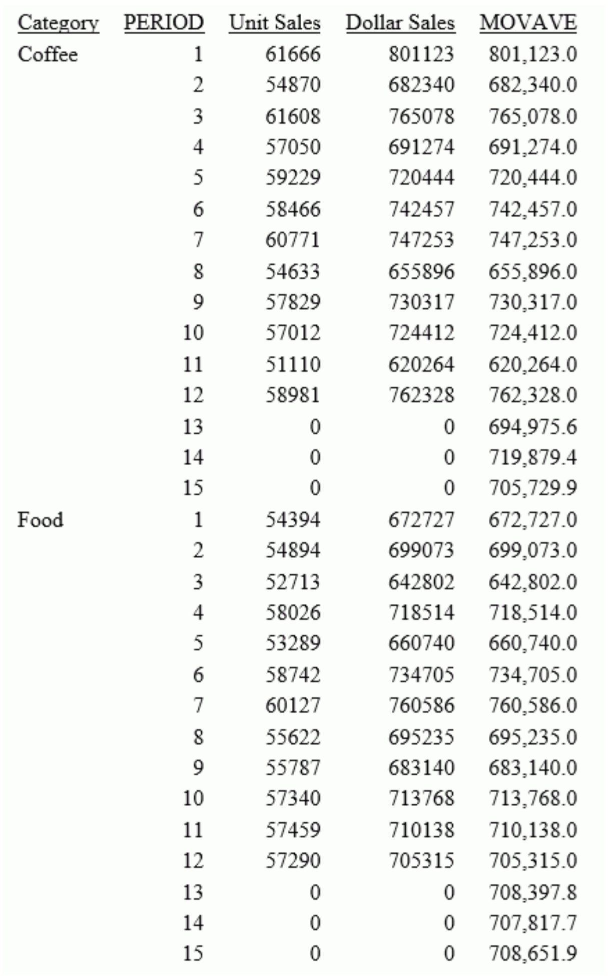
For predicted values beyond the supplied values, the calculated MOVAVE values are used as new data points to continue the moving average. The predicted MOVAVE values (starting with 694,975.6 for PERIOD 13) are calculated using the previous MOVAVE values as new data points. For example, the first predicted value (694,975.6) is the average of the data points from periods 11 and 12 (620,264 and 762,328) and the moving average for period 12 (702,334.7). The calculation is: 694,975 = (620,264 + 762,328 + 702,334.7)/3.

Example: Displaying Original Field Values in a Simple Moving Average Column

This request defines an integer value named PERIOD to use as the independent variable for the moving average. It predicts three periods of values beyond the range of the retrieved data. It uses the keyword INPUT_FIELD as the first argument in the FORECAST parameter list. The trend values do not display in the report. The actual data values for DOLLARS are followed by the predicted values in the report column.

```
DEFINE FILE GGSALES
SDATE/YYM = DATE;
SYEAR/Y = SDATE;
SMONTH/M = SDATE;
PERIOD/I2 = SMONTH;
END
TABLE FILE GGSALES
SUM UNITS DOLLARS
COMPUTE MOVAVE/D10.1 = FORECAST_MOVAVE(INPUT_FIELD,DOLLARS,1,3,3);
BY CATEGORY BY PERIOD
WHERE SYEAR EQ 97 AND CATEGORY NE 'Gifts'
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Category	PERIOD	Unit Sales	Dollar Sales	MOVAVE
Coffee	1	61666	801123	801,123.0
	2	54870	682340	682,340.0
	3	61608	765078	765,078.0
	4	57050	691274	691,274.0
	5	59229	720444	720,444.0
	6	58466	742457	742,457.0
	7	60771	747253	747,253.0
	8	54633	655896	655,896.0
	9	57829	730317	730,317.0
	10	57012	724412	724,412.0
	11	51110	620264	620,264.0
	12	58981	762328	762,328.0
	13	0	0	694,975.6
	14	0	0	719,879.4
	15	0	0	705,729.9
Food	1	54394	672727	672,727.0
	2	54894	699073	699,073.0
	3	52713	642802	642,802.0
	4	58026	718514	718,514.0
	5	53289	660740	660,740.0
	6	58742	734705	734,705.0
	7	60127	760586	760,586.0
	8	55622	695235	695,235.0
	9	55787	683140	683,140.0
	10	57340	713768	713,768.0
	11	57459	710138	710,138.0
	12	57290	705315	705,315.0
	13	0	0	708,397.8
	14	0	0	707,817.7
	15	0	0	708,651.9

FORECAST_EXPAVE: Using Single Exponential Smoothing

The single exponential smoothing method calculates an average that allows you to choose weights to apply to newer and older values.

The following formula determines the weight given to the newest value.

```
k = 2/(1+n)
```

where:

k
: Is the newest value.

n
: Is an integer greater than one. Increasing *n* increases the weight assigned to the earlier observations (or data instances), as compared to the later ones.

The next calculation of the exponential moving average (EMA) value is derived by the following formula:

```
EMA = (EMA * (1-k)) + (datavalue * k)
```

This means that the newest value from the data source is multiplied by the factor *k* and the current moving average is multiplied by the factor (1-*k*). These quantities are then summed to generate the new EMA.

Note: When the data values are exhausted, the last data value in the sort group is used as the next data value.

Syntax: How to Calculate a Single Exponential Smoothing Column

```
FORECAST_EXPAVE(display, infield, interval,
 npredict, npoint1)
```

where:

display
: Keyword

  Specifies which values to display for rows of output that represent existing data. Valid values are:

  - **INPUT_FIELD.** This displays the original field values for rows that represent existing data.
  - **MODEL_DATA.** This displays the calculated values for rows that represent existing data.

Note: You can show both types of output for any field by creating two independent COMPUTE commands in the same request, each with a different display option.

infield

Is any numeric field. It can be the same field as the result field, or a different field. It cannot be a date-time field or a numeric field with date display options.

interval

Is the increment to add to each sort field value (after the last data point) to create the next value. This must be a positive integer. To sort in descending order, use the BY HIGHEST phrase. The result of adding this number to the sort field values is converted to the same format as the sort field.

For date fields, the minimal component in the format determines how the number is interpreted. For example, if the format is YMD, MDY, or DMY, an interval value of 2 is interpreted as meaning two days. If the format is YM, the 2 is interpreted as meaning two months.

npredict

Is the number of predictions for FORECAST to calculate. It must be an integer greater than or equal to zero. Zero indicates that you do not want predictions, and is only supported with a non-recursive FORECAST.

npoint1

For EXPAVE, this number is used to calculate the weights for each component in the average. This value must be a positive whole number. The weight, k, is calculated by the following formula:

`k=2/(1+`*`npoint1`*`)`

Example: Calculating a Single Exponential Smoothing Column

The following defines an integer value named PERIOD to use as the independent variable for the moving average. It predicts three periods of values beyond the range of retrieved data.

```
DEFINE FILE GGSALES
SDATE/YYM = DATE;
SYEAR/Y = SDATE;
SMONTH/M = SDATE;
PERIOD/I2 = SMONTH;
END
TABLE FILE GGSALES
SUM UNITS DOLLARS
COMPUTE EXPAVE/D10.1= FORECAST_EXPAVE(MODEL_DATA,DOLLARS,1,3,3);
BY CATEGORY BY PERIOD
WHERE SYEAR EQ 97 AND CATEGORY NE 'Gifts'
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:

```
Category     PERIOD  Unit Sales  Dollar Sales        EXPAVE
--------     ------  ----------  ------------        ------
Coffee            1       61666        801123     801,123.0
                  2       54870        682340     741,731.5
                  3       61608        765078     753,404.8
                  4       57050        691274     722,339.4
                  5       59229        720444     721,391.7
                  6       58466        742457     731,924.3
                  7       60771        747253     739,588.7
                  8       54633        655896     697,742.3
                  9       57829        730317     714,029.7
                 10       57012        724412     719,220.8
                 11       51110        620264     669,742.4
                 12       58981        762328     716,035.2
                 13           0             0     739,181.6
                 14           0             0     750,754.8
                 15           0             0     756,541.4
Food              1       54394        672727     672,727.0
                  2       54894        699073     685,900.0
                  3       52713        642802     664,351.0
                  4       58026        718514     691,432.5
                  5       53289        660740     676,086.3
                  6       58742        734705     705,395.6
                  7       60127        760586     732,990.8
                  8       55622        695235     714,112.9
                  9       55787        683140     698,626.5
                 10       57340        713768     706,197.2
                 11       57459        710138     708,167.6
                 12       57290        705315     706,741.3
                 13           0             0     706,028.2
                 14           0             0     705,671.6
                 15           0             0     705,493.3
```

In the report, three predicted values of EXPAVE are calculated within each value of CATEGORY. For values outside the range of the data, new PERIOD values are generated by adding the interval value (1) to the prior PERIOD value.

Each average (EXPAVE value) is computed using DOLLARS values where they exist. The calculation of the moving average begins in the following way:

- ❏ The first EXPAVE value (801,123.0) is the same as the first DOLLARS value.
- ❏ The second EXPAVE value (741,731.5) is calculated as follows. Note that because of rounding and the number of decimal places used, the value derived in this sample calculation varies slightly from the one displayed in the report output:

  ```
  n=3 (number used to calculate weights)

  k = 2/(1+n) = 2/4 = 0.5

  EXPAVE = (EXPAVE*(1-k))+(new-DOLLARS*k) = (801123*0.5) + (682340*0.50) =
  400561.5 + 341170 = 741731.5
  ```

- The third EXPAVE value (753,404.8) is calculated as follows:

  ```
  EXPAVE = (EXPAVE*(1-k))+(new-DOLLARS*k) = (741731.5*0.5)+(765078*0.50) =
  370865.75 + 382539 = 753404.75
  ```

FORECAST_DOUBLEXP: Using Double Exponential Smoothing

Double exponential smoothing produces an exponential moving average that takes into account the tendency of data to either increase or decrease over time without repeating. This is accomplished by using two equations with two constants.

- The first equation accounts for the current time period and is a weighted average of the current data value and the prior average, with an added component (b) that represents the trend for the previous period. The weight constant is k:

  ```
  DOUBLEXP(t) = k * datavalue(t) + (1-k) * ((DOUBLEXP(t-1) + b(t-1))
  ```

- The second equation is the calculated trend value, and is a weighted average of the difference between the current and previous average and the trend for the previous time period. b(*t*) represents the average trend. The weight constant is g:

  ```
  b(t) = g * (DOUBLEXP(t)-DOUBLEXP(t-1)) + (1 - g) * (b(t-1))
  ```

These two equations are solved to derive the smoothed average. The first smoothed average is set to the first data value. The first trend component is set to zero. For choosing the two constants, the best results are usually obtained by minimizing the mean-squared error (MSE) between the data values and the calculated averages. You may need to use nonlinear optimization techniques to find the optimal constants.

The equation used for forecasting beyond the data points with double exponential smoothing is

```
forecast(t+m) = DOUBLEXP(t) + m * b(t)
```

where:

m

Is the number of time periods ahead for the forecast.

Syntax: How to Calculate a Double Exponential Smoothing Column

```
FORECAST_DOUBLEXP(display, infield,
interval, npredict, npoint1, npoint2)
```

where:

display

Keyword

Specifies which values to display for rows of output that represent existing data. Valid values are:

- **INPUT_FIELD.** This displays the original field values for rows that represent existing data.
- **MODEL_DATA.** This displays the calculated values for rows that represent existing data.

Note: You can show both types of output for any field by creating two independent COMPUTE commands in the same request, each with a different display option.

infield

Is any numeric field. It can be the same field as the result field, or a different field. It cannot be a date-time field or a numeric field with date display options.

interval

Is the increment to add to each sort field value (after the last data point) to create the next value. This must be a positive integer. To sort in descending order, use the BY HIGHEST phrase. The result of adding this number to the sort field values is converted to the same format as the sort field.

For date fields, the minimal component in the format determines how the number is interpreted. For example, if the format is YMD, MDY, or DMY, an interval value of 2 is interpreted as meaning two days. If the format is YM, the 2 is interpreted as meaning two months.

npredict

Is the number of predictions for FORECAST to calculate. It must be an integer greater than or equal to zero. Zero indicates that you do not want predictions, and is only supported with a non-recursive FORECAST.

npoint1

For DOUBLEXP, this number is used to calculate the weights for each component in the average. This value must be a positive whole number. The weight, k, is calculated by the following formula:

```
k=2/(1+npoint1)
```

npoint2

For DOUBLEXP, this positive whole number is used to calculate the weights for each term in the trend. The weight, g, is calculated by the following formula:

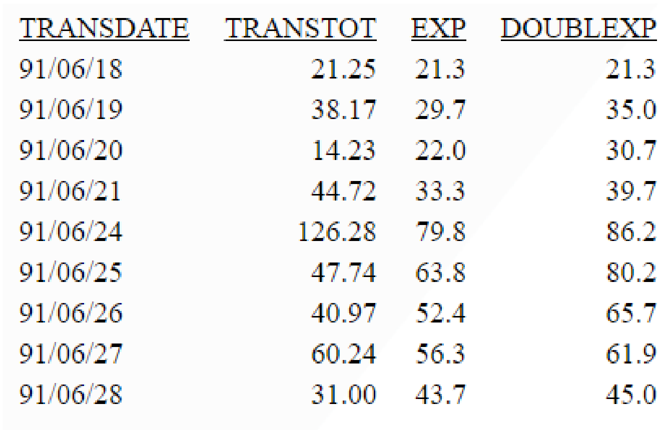
```
g=2/(1+npoint2)
```

Example: **Calculating a Double Exponential Smoothing Column**

The following sums the TRANSTOT field of the VIDEOTRK data source by TRANSDATE, and calculates a single exponential and double exponential moving average. The report columns show the calculated values for existing data points.

```
TABLE FILE VIDEOTRK
SUM TRANSTOT
COMPUTE EXP/D15.1 = FORECAST_EXPAVE(MODEL_DATA,TRANSTOT,1,0,3);
DOUBLEXP/D15.1 = FORECAST_DOUBLEXP(MODEL_DATA,TRANSTOT,1,0,3,3);
BY TRANSDATE
WHERE TRANSDATE NE '19910617'
ON TABLE SET STYLE *
GRID=OFF,$
END
```

The output is shown in the following image:



TRANSDATE	TRANSTOT	EXP	DOUBLEXP
91/06/18	21.25	21.3	21.3
91/06/19	38.17	29.7	35.0
91/06/20	14.23	22.0	30.7
91/06/21	44.72	33.3	39.7
91/06/24	126.28	79.8	86.2
91/06/25	47.74	63.8	80.2
91/06/26	40.97	52.4	65.7
91/06/27	60.24	56.3	61.9
91/06/28	31.00	43.7	45.0

FORECAST_SEASONAL: Using Triple Exponential Smoothing

Triple exponential smoothing produces an exponential moving average that takes into account the tendency of data to repeat itself in intervals over time. For example, sales data that is growing and in which 25% of sales always occur during December contains both trend and seasonality. Triple exponential smoothing takes both the trend and seasonality into account by using three equations with three constants.

For triple exponential smoothing you, need to know the number of data points in each time period (designated as L in the following equations). To account for the seasonality, a seasonal index is calculated. The data is divided by the prior season index and then used in calculating the smoothed average.

- The first equation accounts for the current time period, and is a weighted average of the current data value divided by the seasonal factor and the prior average adjusted for the trend for the previous period. The weight constant is k:

  ```
  SEASONAL(t) = k * (datavalue(t)/I(t-L)) + (1-k) * (SEASONAL(t-1) +
  b(t-1))
  ```

- The second equation is the calculated trend value, and is a weighted average of the difference between the current and previous average and the trend for the previous time period. b(*t*) represents the average trend. The weight constant is g:

  ```
  b(t) = g * (SEASONAL(t)-SEASONAL(t-1)) + (1-g) * (b(t-1))
  ```

- The third equation is the calculated seasonal index, and is a weighted average of the current data value divided by the current average and the seasonal index for the previous season. I(*t*) represents the average seasonal coefficient. The weight constant is p:

  ```
  I(t) = p * (datavalue(t)/SEASONAL(t)) + (1 - p) * I(t-L)
  ```

These equations are solved to derive the triple smoothed average. The first smoothed average is set to the first data value. Initial values for the seasonality factors are calculated based on the maximum number of full periods of data in the data source, while the initial trend is calculated based on two periods of data. These values are calculated with the following steps:

1. The initial trend factor is calculated by the following formula:

   ```
   b(0) = (1/L) ((y(L+1)-y(1))/L + (y(L+2)-y(2))/L + ... + (y(2L) -
   y(L))/L )
   ```

2. The calculation of the initial seasonality factor is based on the average of the data values within each period, A(j) (1<=j<=N):

   ```
   A(j) = ( y((j-1)L+1) + y((j-1)L+2) + ... + y(jL) ) / L
   ```

3. Then, the initial periodicity factor is given by the following formula, where N is the number of full periods available in the data, L is the number of points per period and n is a point within the period (1<= n <= L):

   ```
   I(n) = ( y(n)/A(1) + y(L+n)/A(2) + ... + y((N-1)L+n)/A(N) ) / N
   ```

The three constants must be chosen carefully. The best results are usually obtained by choosing the constants to minimize the mean-squared error (MSE) between the data values and the calculated averages. Varying the values of npoint1 and npoint2 affect the results, and some values may produce a better approximation. To search for a better approximation, you may want to find values that minimize the MSE.

The equation used to forecast beyond the last data point with triple exponential smoothing is:

```
forecast(t+m) = (SEASONAL(t) + m * b(t)) / I(t-L+MOD(m/L))
```

where:

m

Is the number of periods ahead for the forecast.

Syntax: How to Calculate a Triple Exponential Smoothing Column

```
FORECAST_SEASONAL(display, infield,
interval, npredict, nperiod, npoint1, npoint2, npoint3)
```

where:

display

Keyword

Specifies which values to display for rows of output that represent existing data. Valid values are:

- **INPUT_FIELD.** This displays the original field values for rows that represent existing data.
- **MODEL_DATA.** This displays the calculated values for rows that represent existing data.

Note: You can show both types of output for any field by creating two independent COMPUTE commands in the same request, each with a different display option.

infield

Is any numeric field. It can be the same field as the result field, or a different field. It cannot be a date-time field or a numeric field with date display options.

interval

Is the increment to add to each sort field value (after the last data point) to create the next value. This must be a positive integer. To sort in descending order, use the BY HIGHEST phrase. The result of adding this number to the sort field values is converted to the same format as the sort field.

For date fields, the minimal component in the format determines how the number is interpreted. For example, if the format is YMD, MDY, or DMY, an interval value of 2 is interpreted as meaning two days. If the format is YM, the 2 is interpreted as meaning two months.

npredict

Is the number of predictions for FORECAST to calculate. It must be an integer greater than or equal to zero. Zero indicates that you do not want predictions, and is only supported with a non-recursive FORECAST. For the SEASONAL method, npredict is the number of *periods* to calculate. The number of *points* generated is:

```
nperiod * npredict
```

nperiod

For the SEASONAL method, is a positive whole number that specifies the number of data points in a period.

npoint1

For SEASONAL, this number is used to calculate the weights for each component in the average. This value must be a positive whole number. The weight, k, is calculated by the following formula:

```
k=2/(1+npoint1)
```

npoint2

For SEASONAL, this positive whole number is used to calculate the weights for each term in the trend. The weight, g, is calculated by the following formula:

```
g=2/(1+npoint2)
```

npoint3

For SEASONAL, this positive whole number is used to calculate the weights for each term in the seasonal adjustment. The weight, p, is calculated by the following formula:

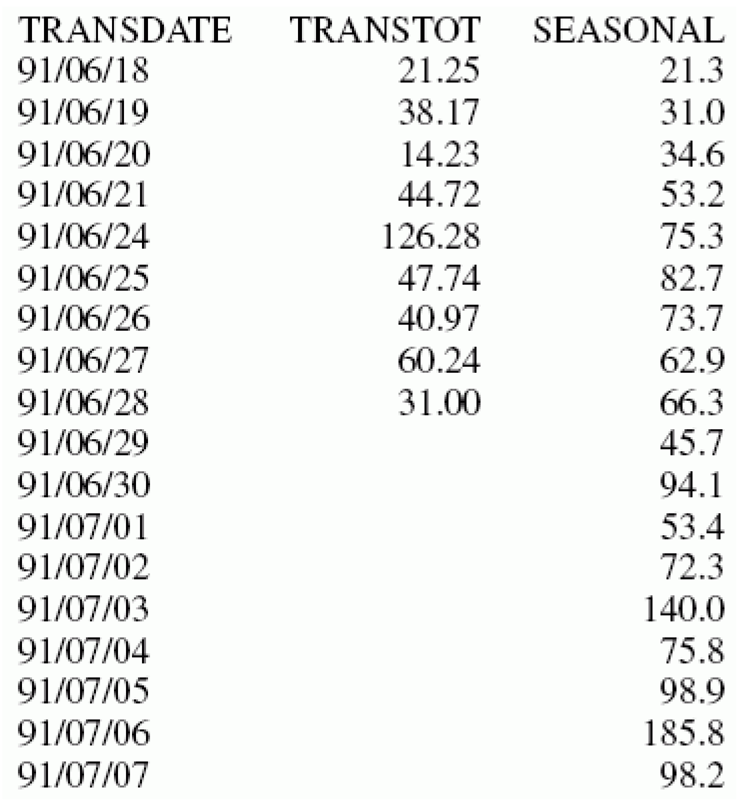
```
p=2/(1+npoint3)
```

Example: Calculating a Triple Exponential Smoothing Column

In the following, the data has seasonality but no trend. Therefore, *npoint2* is set high (1000) to make the trend factor negligible in the calculation:

```
TABLE FILE VIDEOTRK
SUM TRANSTOT
COMPUTE SEASONAL/D10.1 = FORECAST_SEASONAL(MODEL_DATA,TRANSTOT,
1,3,3,3,1000,1);
BY TRANSDATE
WHERE TRANSDATE NE '19910617'
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

In the output, *npredict* is 3. Therefore, three periods (nine points, *nperiod* * *npredict*) are generated.



```
TRANSDATE    TRANSTOT   SEASONAL
91/06/18        21.25       21.3
91/06/19        38.17       31.0
91/06/20        14.23       34.6
91/06/21        44.72       53.2
91/06/24       126.28       75.3
91/06/25        47.74       82.7
91/06/26        40.97       73.7
91/06/27        60.24       62.9
91/06/28        31.00       66.3
91/06/29                    45.7
91/06/30                    94.1
91/07/01                    53.4
91/07/02                    72.3
91/07/03                   140.0
91/07/04                    75.8
91/07/05                    98.9
91/07/06                   185.8
91/07/07                    98.2
```

FORECAST_LINEAR: Using a Linear Regression Equation

The linear regression equation estimates values by assuming that the dependent variable (the new calculated values) and the independent variable (the sort field values) are related by a function that represents a straight line:

$y = mx + b$

where:

y

Is the dependent variable.

x

Is the independent variable.

m

Is the slope of the line.

b

Is the y-intercept.

FORECAST_LINEAR uses a technique called Ordinary Least Squares to calculate values for *m* and *b* that minimize the sum of the squared differences between the data and the resulting line.

The following formulas show how *m* and *b* are calculated.

$$m = \frac{(\sum xy - (\sum x \bullet \sum y)/n)}{(\sum x^2 - (\sum x)^2/n)}$$

$$b = (\sum y)/n - (m \bullet (\sum x)/n)$$

where:

n

Is the number of data points.

y

Is the data values (dependent variables).

x

Is the sort field values (independent variables).

Trend values, as well as predicted values, are calculated using the regression line equation.

Syntax: How to Calculate a Linear Regression Column

```
FORECAST_LINEAR(display, infield, interval,
 npredict)
```

where:

display

Keyword

Specifies which values to display for rows of output that represent existing data. Valid values are:

- **INPUT_FIELD.** This displays the original field values for rows that represent existing data.
- **MODEL_DATA.** This displays the calculated values for rows that represent existing data.

Note: You can show both types of output for any field by creating two independent COMPUTE commands in the same request, each with a different display option.

infield

Is any numeric field. It can be the same field as the result field, or a different field. It cannot be a date-time field or a numeric field with date display options.

interval

Is the increment to add to each sort field value (after the last data point) to create the next value. This must be a positive integer. To sort in descending order, use the BY HIGHEST phrase. The result of adding this number to the sort field values is converted to the same format as the sort field.

For date fields, the minimal component in the format determines how the number is interpreted. For example, if the format is YMD, MDY, or DMY, an interval value of 2 is interpreted as meaning two days. If the format is YM, the 2 is interpreted as meaning two months.

npredict

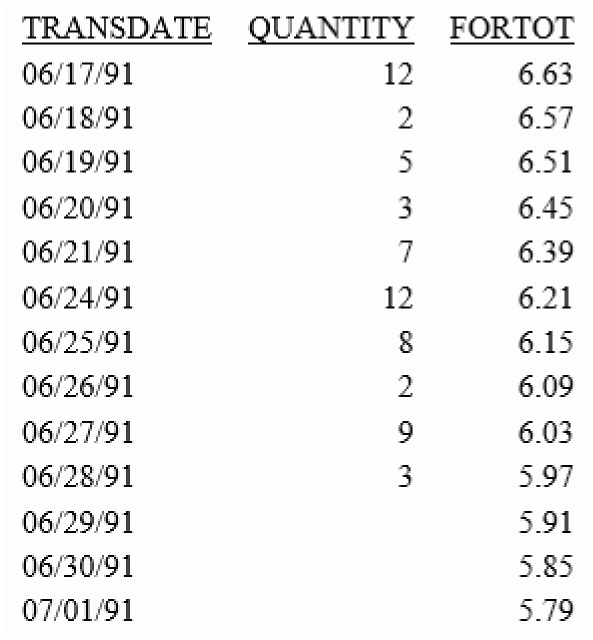
Is the number of predictions for FORECAST to calculate. It must be an integer greater than or equal to zero. Zero indicates that you do not want predictions, and is only supported with a non-recursive FORECAST.

Example: Calculating a New Linear Regression Field

The following request calculates a regression line using the VIDEOTRK data source of QUANTITY by TRANSDATE. The interval is one day, and three predicted values are calculated.

```
TABLE FILE VIDEOTRK
SUM QUANTITY
COMPUTE FORTOT=FORECAST_LINEAR(MODEL_DATA,QUANTITY,1,3);
BY TRANSDATE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



TRANSDATE	QUANTITY	FORTOT
06/17/91	12	6.63
06/18/91	2	6.57
06/19/91	5	6.51
06/20/91	3	6.45
06/21/91	7	6.39
06/24/91	12	6.21
06/25/91	8	6.15
06/26/91	2	6.09
06/27/91	9	6.03
06/28/91	3	5.97
06/29/91		5.91
06/30/91		5.85
07/01/91		5.79

Note:

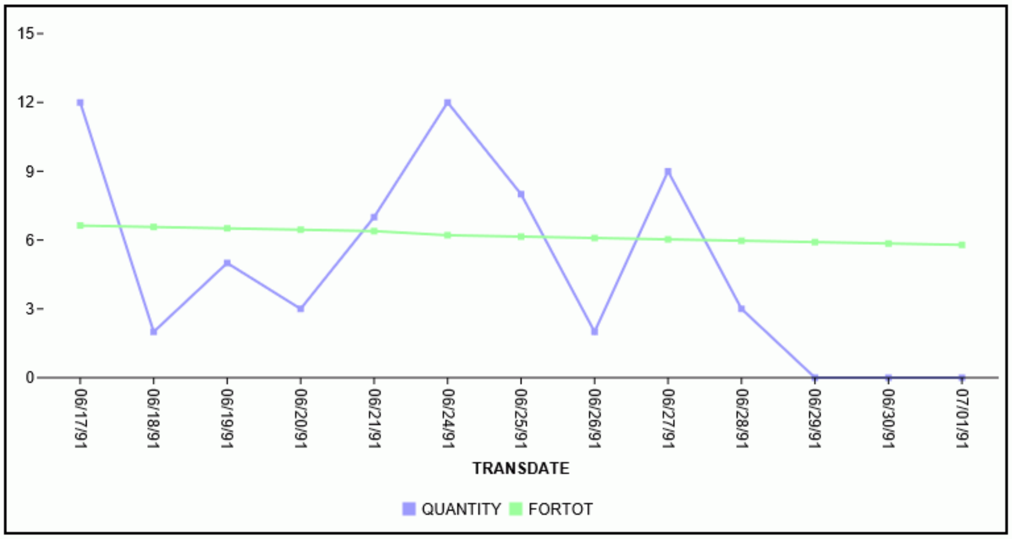
- Three predicted values of FORTOT are calculated. For values outside the range of the data, new TRANSDATE values are generated by adding the interval value (1) to the prior TRANSDATE value.
- There are no QUANTITY values for the generated FORTOT values.
- Each FORTOT value is computed using a regression line, calculated using all of the actual data values for QUANTITY.

TRANSDATE is the independent variable (x) and QUANTITY is the dependent variable (y). The equation is used to calculate QUANTITY FORECAST trend and predicted values.

The following version of the request charts the data values and the regression line.

```
GRAPH FILE VIDEOTRK
SUM QUANTITY
COMPUTE FORTOT=FORECAST_LINEAR(MODEL_DATA,QUANTITY,1,3);
BY TRANSDATE
ON GRAPH HOLD FORMAT JSCHART
ON GRAPH SET LOOKGRAPH VLINE
END
```

The output is shown in the following image.



PARTITION_AGGR: Creating Rolling Calculations

Using the PARTITION_AGGR function, you can generate rolling calculations based on a block of rows from the internal matrix of a TABLE request. In order to determine the limits of the rolling calculations, you specify a partition of the data based on either a sort field or the entire TABLE. Within either type of break, you can start calculating from the beginning of the break or a number of rows prior to or subsequent to the current row. You can stop the rolling calculation at the current row, a row past the start point, or the end of the partition.

By default, the field values used in the calculations are the summed values of a measure in the request. Certain prefix operators can be used to add a column to the internal matrix and use that column in the rolling calculations. The rolling calculation can be SUM, AVE, CNT, MIN, MAX, FST, or LST.

Syntax: How to Generate Rolling Calculations Using PARTITION_AGGR

```
PARTITION_AGGR([prefix.]measure, reset_key, lower, upper, operation)
```

where:

prefix.

Defines an aggregation operator to apply to the measure before using it in the rolling calculation. Valid operators are:

- **SUM.** which calculates the sum of the measure field values. SUM is the default operator.
- **CNT.** which calculates a count of the measure field values.
- **AVE.** which calculates the average of the measure field values.
- **MIN.** which calculates the minimum of the measure field values.
- **MAX.** which calculates the maximum of the measure field values.
- **FST.** which retrieves the first value of the measure field.
- **LST.** which retrieves the last value of the measure field.
- **STDP.** which calculates the population standard deviation.
- **STDS.** which calculates the sample standard deviation.

Note: The operators PCT., RPCT., TOT., MDN., and DST. are not supported. COMPUTEs that reference those unsupported operators are also not supported.

measure

Is the measure field to be aggregated. It can be a real field in the request or a calculated value generated with the COMPUTE command, as long as the COMPUTE does not reference an unsupported prefix operator.

reset_key

Identifies the point at which the calculation restarts. Valid values are:

- The name of a sort field in the request.

- PRESET, which uses the value of the PARTITION_ON parameter, as described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.
- TABLE, which indicates that there is no break on a sort field.

The sort field may use BY HIGHEST to indicate a HIGH-TO-LOW sort. ACROSS COLUMNS AND is supported. BY ROWS OVER and FOR are not supported.

lower

Identifies the starting point for the rolling calculation. Valid values are:

- **n, -n**, which starts the calculation *n* rows forward or back from the current row.
- **B**, which starts the calculation at the beginning of the current sort break (the first line with the same sort field value as the current line).

upper

Identifies the ending point of the rolling calculation. The *lower* row value must precede *upper* row value.

Valid values are:

- **C**, which ends the rolling calculation at the current row in the internal matrix.
- **n, -n**, which ends the calculation *n* rows forward or back from the current row.
- **E**, which ends the rolling calculation at the end of the sort break (the last line with the same sort value as the current row.)

Note: The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

operation

Specifies the rolling calculation used on the values in the internal matrix. Supported operations are:

- **SUM,** which calculates a rolling sum.
- **AVE,** which calculates a rolling average.
- **CNT,** which counts the rows in the partition.
- MEDIAN.
- **MIN,** which returns the minimum value in the partition.
- **MAX,** which returns the maximum value in the partition.

- **MEDIAN,** which returns the median value in the partition.
- **MODE,** which returns the mode value in the partition.
- **FST,** which returns the first value in the partition.
- **LST,** which returns the last value in the partition.
- **STDP,** which returns the population standard deviation in the partition. Requires using the verb PRINT to avoid duplicate aggregation.
- **STDS,** which returns the sample standard deviation in the partition. Requires using the verb PRINT to avoid duplicate aggregation.

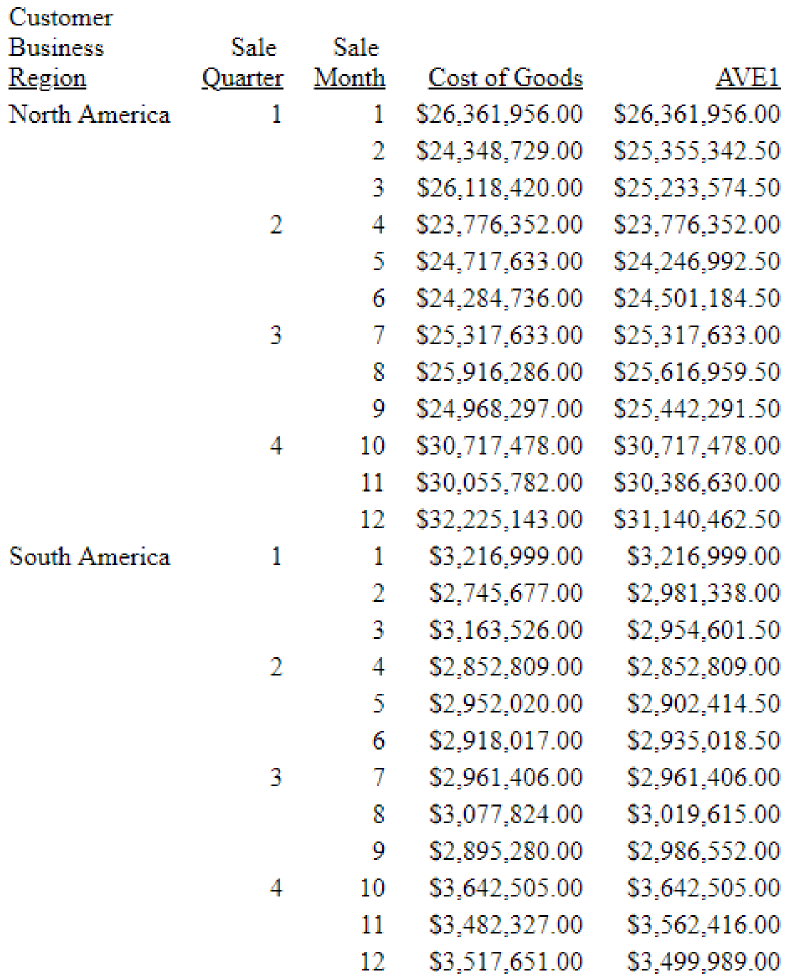
The calculation is performed prior to any WHERE TOTAL tests, but after any WHERE_GROUPED tests.

Example: Calculating a Rolling Average

The following request calculates a rolling average of the current line and the previous line in the internal matrix, within the quarter.

```
TABLE FILE WF_RETAIL_LITE
SUM COGS_US
COMPUTE AVE1/D12.2M = PARTITION_AGGR(COGS_US, TIME_QTR, -1, C, AVE);
BY BUSINESS_REGION
BY TIME_QTR
BY TIME_MTH
WHERE BUSINESS_REGION EQ 'North America' OR 'South America'
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. Within each quarter, the first average is just the value from Q1, as going back 1 would cross a boundary. The second average is calculated using the first two rows within that quarter, and the third average is calculated using rows 2 and 3 within the quarter.

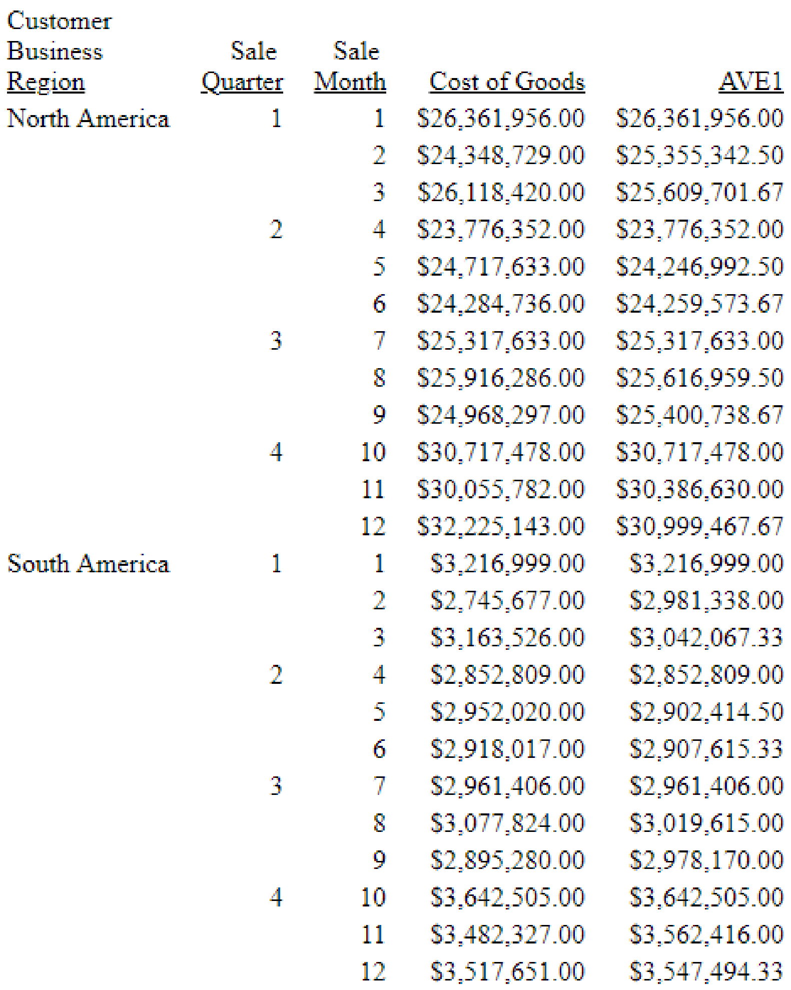


Customer Business Region	Sale Quarter	Sale Month	Cost of Goods	AVE1
North America	1	1	$26,361,956.00	$26,361,956.00
		2	$24,348,729.00	$25,355,342.50
		3	$26,118,420.00	$25,233,574.50
	2	4	$23,776,352.00	$23,776,352.00
		5	$24,717,633.00	$24,246,992.50
		6	$24,284,736.00	$24,501,184.50
	3	7	$25,317,633.00	$25,317,633.00
		8	$25,916,286.00	$25,616,959.50
		9	$24,968,297.00	$25,442,291.50
	4	10	$30,717,478.00	$30,717,478.00
		11	$30,055,782.00	$30,386,630.00
		12	$32,225,143.00	$31,140,462.50
South America	1	1	$3,216,999.00	$3,216,999.00
		2	$2,745,677.00	$2,981,338.00
		3	$3,163,526.00	$2,954,601.50
	2	4	$2,852,809.00	$2,852,809.00
		5	$2,952,020.00	$2,902,414.50
		6	$2,918,017.00	$2,935,018.50
	3	7	$2,961,406.00	$2,961,406.00
		8	$3,077,824.00	$3,019,615.00
		9	$2,895,280.00	$2,986,552.00
	4	10	$3,642,505.00	$3,642,505.00
		11	$3,482,327.00	$3,562,416.00
		12	$3,517,651.00	$3,499,989.00

The following changes the rolling average to start from the beginning of the sort break.

```
COMPUTE AVE1/D12.2M = PARTITION_AGGR(COGS_US, TIME_QTR ,B, C, AVE);
```

The output is shown in the following image. Within each quarter, the first average is just the value from Q1, as going back would cross a boundary. The second average is calculated using the first two rows within that quarter, and the third average is calculated using rows 1 through 3 within the quarter.

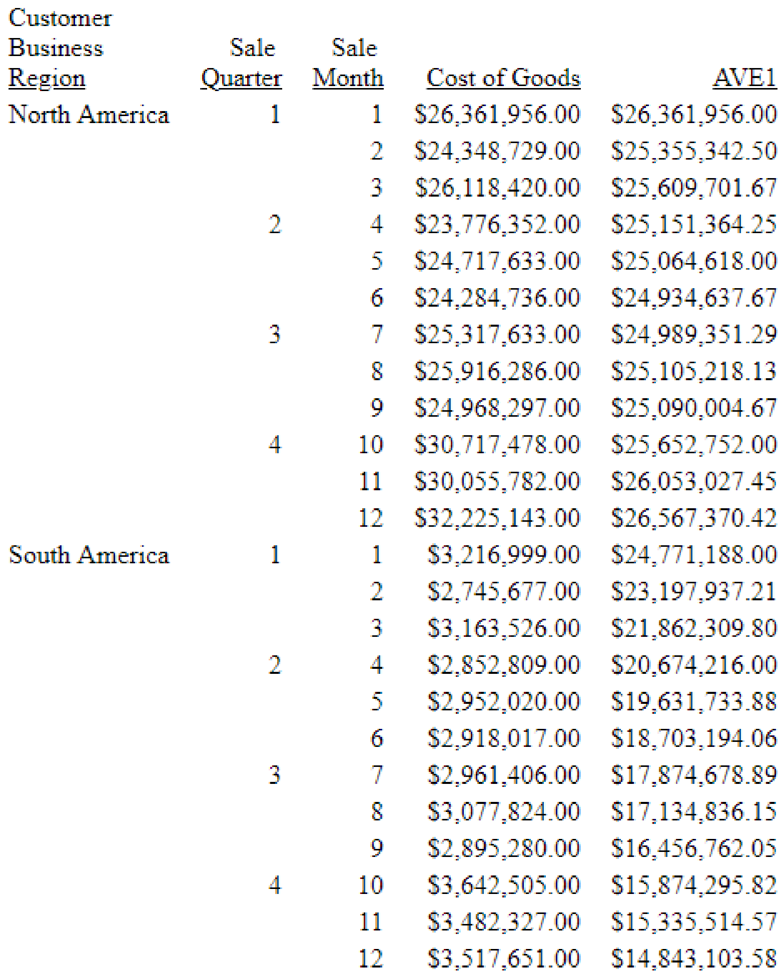


Customer Business Region	Sale Quarter	Sale Month	Cost of Goods	AVE1
North America	1	1	$26,361,956.00	$26,361,956.00
		2	$24,348,729.00	$25,355,342.50
		3	$26,118,420.00	$25,609,701.67
	2	4	$23,776,352.00	$23,776,352.00
		5	$24,717,633.00	$24,246,992.50
		6	$24,284,736.00	$24,259,573.67
	3	7	$25,317,633.00	$25,317,633.00
		8	$25,916,286.00	$25,616,959.50
		9	$24,968,297.00	$25,400,738.67
	4	10	$30,717,478.00	$30,717,478.00
		11	$30,055,782.00	$30,386,630.00
		12	$32,225,143.00	$30,999,467.67
South America	1	1	$3,216,999.00	$3,216,999.00
		2	$2,745,677.00	$2,981,338.00
		3	$3,163,526.00	$3,042,067.33
	2	4	$2,852,809.00	$2,852,809.00
		5	$2,952,020.00	$2,902,414.50
		6	$2,918,017.00	$2,907,615.33
	3	7	$2,961,406.00	$2,961,406.00
		8	$3,077,824.00	$3,019,615.00
		9	$2,895,280.00	$2,978,170.00
	4	10	$3,642,505.00	$3,642,505.00
		11	$3,482,327.00	$3,562,416.00
		12	$3,517,651.00	$3,547,494.33

The following command uses the partition boundary TABLE.

```
COMPUTE AVE1/D12.2M = PARTITION_AGGR(COGS_US, TABLE, B, C, AVE);
```

The output is shown in the following image. The rolling average keeps adding the next row to the average with no sort field break.



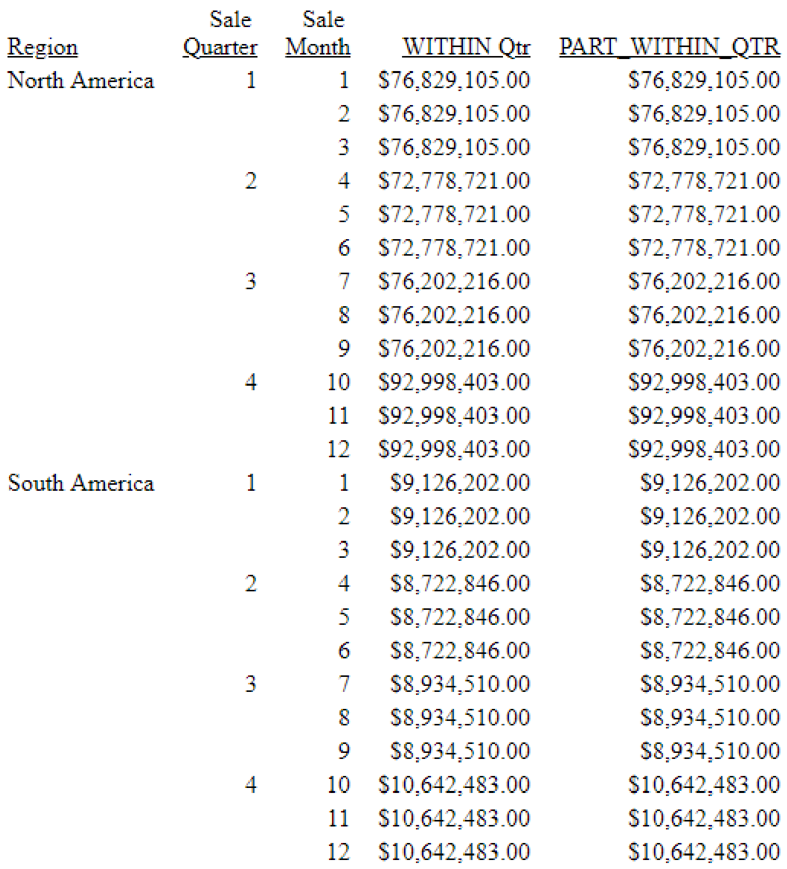
```
Customer
Business             Sale     Sale
Region            Quarter    Month   Cost of Goods             AVE1
North America           1        1  $26,361,956.00   $26,361,956.00
                                 2  $24,348,729.00   $25,355,342.50
                                 3  $26,118,420.00   $25,609,701.67
                        2        4  $23,776,352.00   $25,151,364.25
                                 5  $24,717,633.00   $25,064,618.00
                                 6  $24,284,736.00   $24,934,637.67
                        3        7  $25,317,633.00   $24,989,351.29
                                 8  $25,916,286.00   $25,105,218.13
                                 9  $24,968,297.00   $25,090,004.67
                        4       10  $30,717,478.00   $25,652,752.00
                                11  $30,055,782.00   $26,053,027.45
                                12  $32,225,143.00   $26,567,370.42
South America           1        1   $3,216,999.00   $24,771,188.00
                                 2   $2,745,677.00   $23,197,937.21
                                 3   $3,163,526.00   $21,862,309.80
                        2        4   $2,852,809.00   $20,674,216.00
                                 5   $2,952,020.00   $19,631,733.88
                                 6   $2,918,017.00   $18,703,194.06
                        3        7   $2,961,406.00   $17,874,678.89
                                 8   $3,077,824.00   $17,134,836.15
                                 9   $2,895,280.00   $16,456,762.05
                        4       10   $3,642,505.00   $15,874,295.82
                                11   $3,482,327.00   $15,335,514.57
                                12   $3,517,651.00   $14,843,103.58
```

Reference: Usage Notes for PARTITION_AGGR

- ❑ Fields referenced in the PARTITION_AGGR parameters but not previously mentioned in the request will *not* be counted in column notation or propagated to HOLD files.
- ❑ Using the WITHIN phrase for a sum is the same as computing PARTITION_AGGR on the WITHIN sort field from B (beginning of sort break) to E (end of sort break) using SUM, as in the following example.

```
TABLE FILE WF_RETAIL_LITE
SUM COGS_US WITHIN TIME_QTR AS 'WITHIN Qtr'
COMPUTE PART_WITHIN_QTR/D12.2M = PARTITION_AGGR(COGS_US, TIME_QTR, B, E,
SUM);
BY BUSINESS_REGION AS Region
BY TIME_QTR
BY TIME_MTH
WHERE BUSINESS_REGION EQ 'North America' OR 'South America'
ON TABLE SET PAGE NOPAGE
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image.

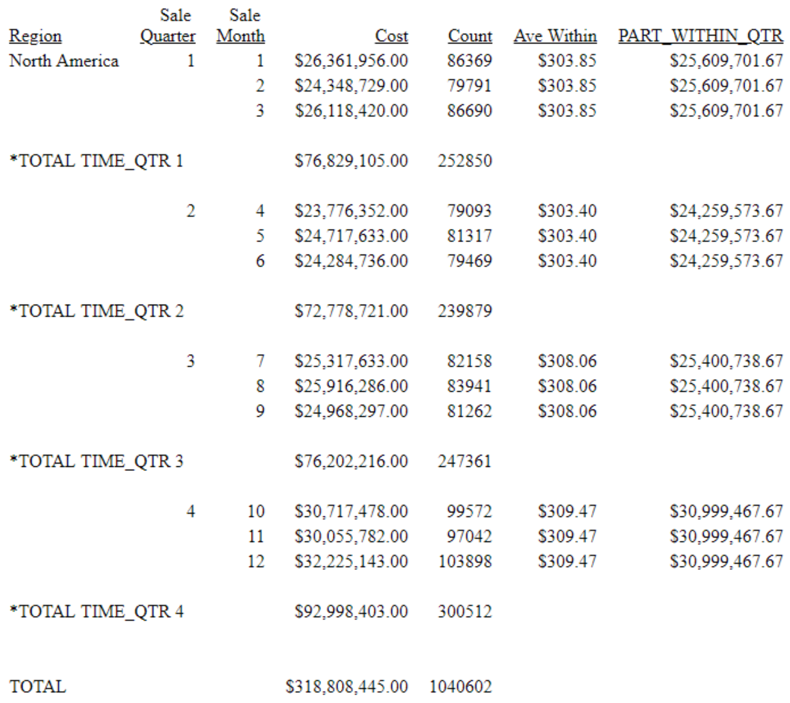


```
                   Sale     Sale
Region           Quarter   Month      WITHIN Qtr   PART_WITHIN_QTR
North America          1       1  $76,829,105.00    $76,829,105.00
                               2  $76,829,105.00    $76,829,105.00
                               3  $76,829,105.00    $76,829,105.00
                       2       4  $72,778,721.00    $72,778,721.00
                               5  $72,778,721.00    $72,778,721.00
                               6  $72,778,721.00    $72,778,721.00
                       3       7  $76,202,216.00    $76,202,216.00
                               8  $76,202,216.00    $76,202,216.00
                               9  $76,202,216.00    $76,202,216.00
                       4      10  $92,998,403.00    $92,998,403.00
                              11  $92,998,403.00    $92,998,403.00
                              12  $92,998,403.00    $92,998,403.00
South America          1       1   $9,126,202.00     $9,126,202.00
                               2   $9,126,202.00     $9,126,202.00
                               3   $9,126,202.00     $9,126,202.00
                       2       4   $8,722,846.00     $8,722,846.00
                               5   $8,722,846.00     $8,722,846.00
                               6   $8,722,846.00     $8,722,846.00
                       3       7   $8,934,510.00     $8,934,510.00
                               8   $8,934,510.00     $8,934,510.00
                               9   $8,934,510.00     $8,934,510.00
                       4      10  $10,642,483.00    $10,642,483.00
                              11  $10,642,483.00    $10,642,483.00
                              12  $10,642,483.00    $10,642,483.00
```

With other types of calculations, the results are not the same. For example, the following request calculates the average within quarter using the WITHIN phrase and the average within quarter using PARTITION_AGGR.

```
TABLE FILE WF_RETAIL_LITE
SUM  COGS_US AS Cost
CNT.COGS_US AS Count AVE.COGS_US WITHIN TIME_QTR AS 'Ave Within'
COMPUTE PART_WITHIN_QTR/D12.2M = PARTITION_AGGR(COGS_US, TIME_QTR, B, E,
AVE);
BY BUSINESS_REGION AS Region
BY TIME_QTR
ON TIME_QTR SUBTOTAL COGS_US CNT.COGS_US
BY TIME_MTH
WHERE BUSINESS_REGION EQ 'North America'
ON TABLE SET PAGE NOPAGE
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. The average using the WITHIN phrase divides the total cost for the quarter by the total count of instances for the quarter (for example, $76,829,105.00/ 252850 = $303.85), while PARTITION_AGGR divides the total cost for the quarter by the number of report rows in the quarter (for example, $76,829,105.00/3 = $25,609,701.67).



```
                   Sale     Sale
Region          Quarter    Month           Cost     Count  Ave Within  PART_WITHIN_QTR
North America         1        1 $26,361,956.00     86369     $303.85   $25,609,701.67
                               2 $24,348,729.00     79791     $303.85   $25,609,701.67
                               3 $26,118,420.00     86690     $303.85   $25,609,701.67

*TOTAL TIME_QTR 1                $76,829,105.00    252850

                      2        4 $23,776,352.00     79093     $303.40   $24,259,573.67
                               5 $24,717,633.00     81317     $303.40   $24,259,573.67
                               6 $24,284,736.00     79469     $303.40   $24,259,573.67

*TOTAL TIME_QTR 2                $72,778,721.00    239879

                      3        7 $25,317,633.00     82158     $308.06   $25,400,738.67
                               8 $25,916,286.00     83941     $308.06   $25,400,738.67
                               9 $24,968,297.00     81262     $308.06   $25,400,738.67

*TOTAL TIME_QTR 3                $76,202,216.00    247361

                      4       10 $30,717,478.00     99572     $309.47   $30,999,467.67
                              11 $30,055,782.00     97042     $309.47   $30,999,467.67
                              12 $32,225,143.00    103898     $309.47   $30,999,467.67

*TOTAL TIME_QTR 4                $92,998,403.00    300512


TOTAL                           $318,808,445.00   1040602
```

- If you use PARTITION_AGGR to perform operations for specific time periods using an offset, for example, an operation on the quarters for different years, you must make sure that every quarter is represented. If some quarters are missing for some years, the offset will not access the correct data. In this case, generate a HOLD file that has every quarter represented for every year (you can use BY QUARTER ROWS OVER 1 OVER 2 OVER 3 OVER 4) and use PARTITION_AGGR on the HOLD file.

PARTITION_REF: Using Prior or Subsequent Field Values in Calculations

Use of LAST in a calculation retrieves the LAST value of the specified field the last time this calculation was performed. In contrast, the PARTITION_REF function enables you to specify both how many rows back or forward to go in the output in order to retrieve a value, and a sort break within which the retrieval will be contained.

Syntax: How to Retrieve Prior or Subsequent Field Values for Use in a Calculation

```
PARTITION_REF([prefix.]field, reset_key, offset)
```

where:

prefix

Is optional. If used, it can be one of the following aggregation operators:

- ❏ **AVE.** Average
- ❏ **MAX.** Maximum
- ❏ **MIN.** Minimum
- ❏ **CNT.** Count
- ❏ **SUM.** Sum

field

Is the field whose value is to be retrieved.

reset_key

Identifies the point at which the retrieval break restarts. Valid values are:

- ❏ The name of a sort field in the request.
- ❏ PRESET, which uses the value of the PARTITION_ON parameter, as described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.
- ❏ TABLE, which indicates that there is no break on a sort field.

The sort field may use BY HIGHEST to indicate a HIGH-TO-LOW sort. ACROSS COLUMNS AND is supported. BY ROWS OVER and FOR are not supported.

Note: The values used in the retrieval depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

offset

Is the integer number of records to go forward (for a positive offset) or backward (for a negative offset) to retrieve the value.

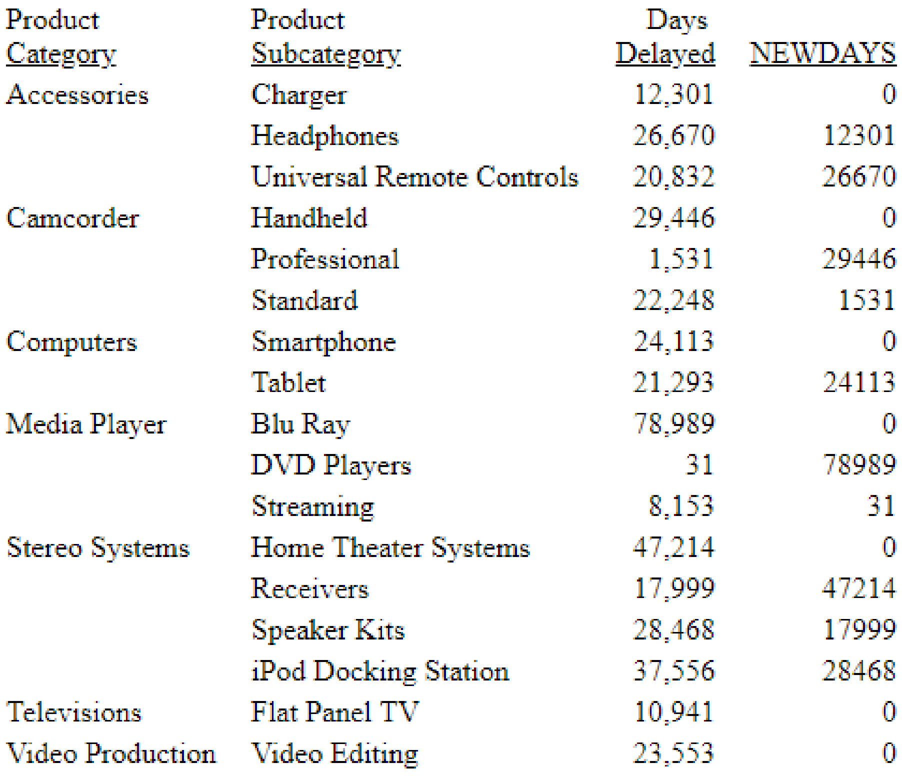
If the offset is prior to the partition boundary sort value, the return will be the default value for the field. The calculation is performed prior to any WHERE TOTAL tests, but after WHERE_GROUPED tests.

Example: Retrieving a Previous Record With PARTITION_REF

The following request retrieves the previous record within the sort field PRODUCT_CATEGORY.

```
TABLE FILE WF_RETAIL_LITE
SUM DAYSDELAYED
COMPUTE NEWDAYS/I5=PARTITION_REF(DAYSDELAYED, PRODUCT_CATEGORY, -1);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOPAGE
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. The first value within each sort break is zero because there is no prior record to retrieve.

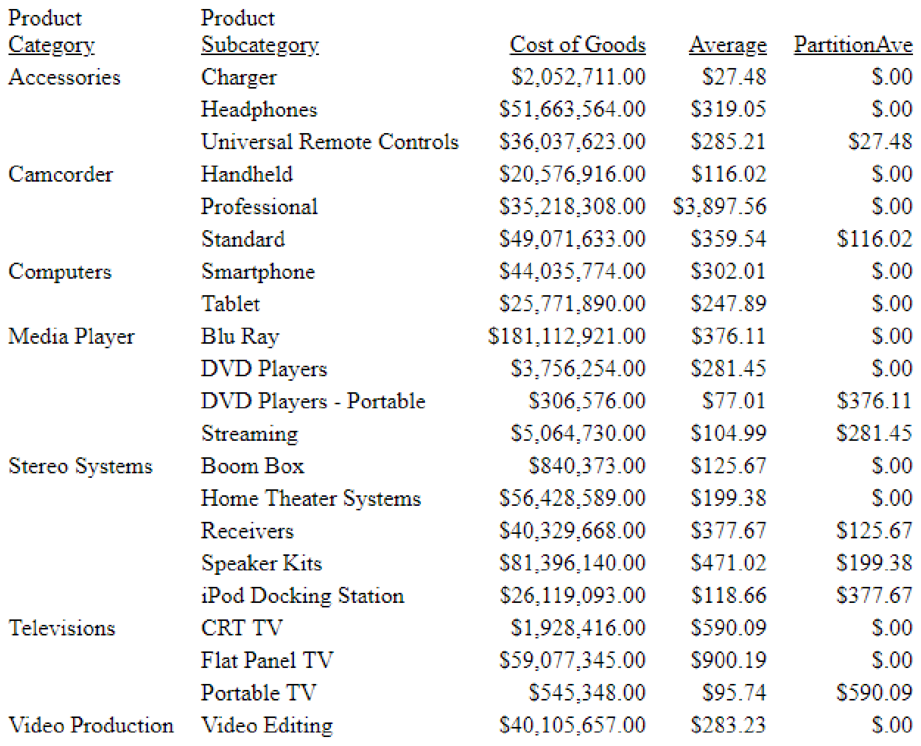


Product Category	Product Subcategory	Days Delayed	NEWDAYS
Accessories	Charger	12,301	0
	Headphones	26,670	12301
	Universal Remote Controls	20,832	26670
Camcorder	Handheld	29,446	0
	Professional	1,531	29446
	Standard	22,248	1531
Computers	Smartphone	24,113	0
	Tablet	21,293	24113
Media Player	Blu Ray	78,989	0
	DVD Players	31	78989
	Streaming	8,153	31
Stereo Systems	Home Theater Systems	47,214	0
	Receivers	17,999	47214
	Speaker Kits	28,468	17999
	iPod Docking Station	37,556	28468
Televisions	Flat Panel TV	10,941	0
Video Production	Video Editing	23,553	0

The following request retrieves the average cost of goods from two records prior to the current record within the PRODUCT_CATEGORY sort field.

```
TABLE FILE WF_RETAIL_LITE
SUM COGS_US AVE.COGS_US AS Average
COMPUTE PartitionAve/D12.2M=PARTITION_REF(AVE.COGS_US, PRODUCT_CATEGORY,
-2);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOPAGE
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image.

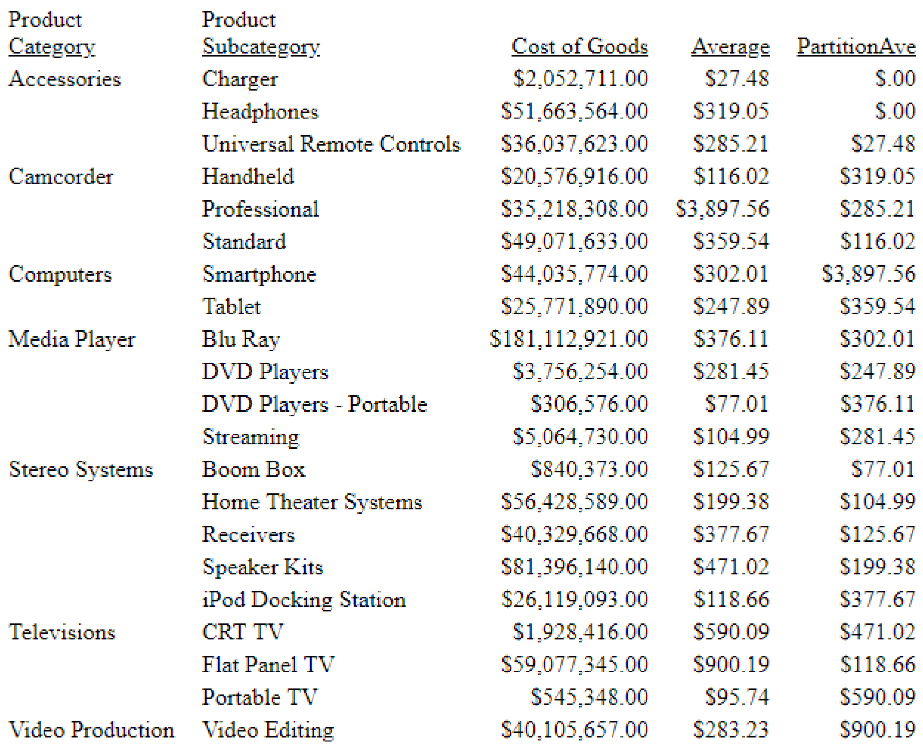


Product Category	Product Subcategory	Cost of Goods	Average	PartitionAve
Accessories	Charger	$2,052,711.00	$27.48	$.00
	Headphones	$51,663,564.00	$319.05	$.00
	Universal Remote Controls	$36,037,623.00	$285.21	$27.48
Camcorder	Handheld	$20,576,916.00	$116.02	$.00
	Professional	$35,218,308.00	$3,897.56	$.00
	Standard	$49,071,633.00	$359.54	$116.02
Computers	Smartphone	$44,035,774.00	$302.01	$.00
	Tablet	$25,771,890.00	$247.89	$.00
Media Player	Blu Ray	$181,112,921.00	$376.11	$.00
	DVD Players	$3,756,254.00	$281.45	$.00
	DVD Players - Portable	$306,576.00	$77.01	$376.11
	Streaming	$5,064,730.00	$104.99	$281.45
Stereo Systems	Boom Box	$840,373.00	$125.67	$.00
	Home Theater Systems	$56,428,589.00	$199.38	$.00
	Receivers	$40,329,668.00	$377.67	$125.67
	Speaker Kits	$81,396,140.00	$471.02	$199.38
	iPod Docking Station	$26,119,093.00	$118.66	$377.67
Televisions	CRT TV	$1,928,416.00	$590.09	$.00
	Flat Panel TV	$59,077,345.00	$900.19	$.00
	Portable TV	$545,348.00	$95.74	$590.09
Video Production	Video Editing	$40,105,657.00	$283.23	$.00

Replacing the function call with the following syntax changes the partition boundary to TABLE.

```
COMPUTE PartitionAve/D12.2M=PARTITION_REF(AVE.COGS_US, TABLE, -2);
```

The output is shown in the following image.



Product Category	Product Subcategory	Cost of Goods	Average	PartitionAve
Accessories	Charger	$2,052,711.00	$27.48	$.00
	Headphones	$51,663,564.00	$319.05	$.00
	Universal Remote Controls	$36,037,623.00	$285.21	$27.48
Camcorder	Handheld	$20,576,916.00	$116.02	$319.05
	Professional	$35,218,308.00	$3,897.56	$285.21
	Standard	$49,071,633.00	$359.54	$116.02
Computers	Smartphone	$44,035,774.00	$302.01	$3,897.56
	Tablet	$25,771,890.00	$247.89	$359.54
Media Player	Blu Ray	$181,112,921.00	$376.11	$302.01
	DVD Players	$3,756,254.00	$281.45	$247.89
	DVD Players - Portable	$306,576.00	$77.01	$376.11
	Streaming	$5,064,730.00	$104.99	$281.45
Stereo Systems	Boom Box	$840,373.00	$125.67	$77.01
	Home Theater Systems	$56,428,589.00	$199.38	$104.99
	Receivers	$40,329,668.00	$377.67	$125.67
	Speaker Kits	$81,396,140.00	$471.02	$199.38
	iPod Docking Station	$26,119,093.00	$118.66	$377.67
Televisions	CRT TV	$1,928,416.00	$590.09	$471.02
	Flat Panel TV	$59,077,345.00	$900.19	$118.66
	Portable TV	$545,348.00	$95.74	$590.09
Video Production	Video Editing	$40,105,657.00	$283.23	$900.19

Reference: Usage Notes for PARTITION_REF

- ❏ Fields referenced in the PARTITION_REF parameters but not previously mentioned in the request, will *not* be counted in column notation or propagated to HOLD files.

INCREASE: Calculating the Difference Between the Current and a Prior Value of a Field

Given an aggregated input field and a negative offset, INCREASE calculates the difference between the value in the current row of the report output and one or more prior rows, within a sort break or the entire table. The reset point for the calculation is determined by the value of the PARTITION_ON parameter described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.

Note: The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

Syntax: How to Calculate the Difference Between the Current and a Prior Value of a Field

```
INCREASE([prefix.]field, offset)
```

where:

prefix

Is one of the following optional aggregation operators to apply to the field before using it in the calculation:

- **SUM.** which calculates the sum of the field values. SUM is the default value.
- **CNT.** which calculates a count of the field values.
- **AVE.** which calculates the average of the field values.
- **MIN.** which calculates the minimum of the field values.
- **MAX.** which calculates the maximum of the field values.
- **FST.** which retrieves the first value of the field.
- **LST.** which retrieves the last value of the field.

field

Numeric

Is the field to be used in the calculation.

offset

Numeric

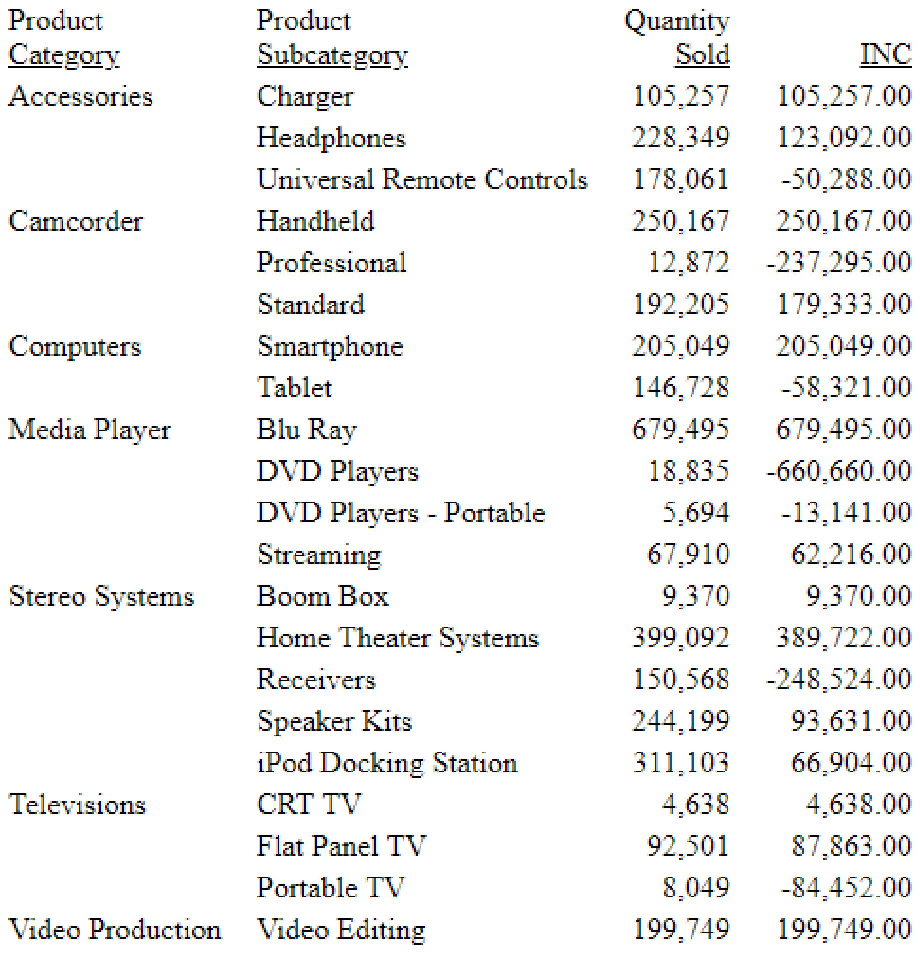
Is a negative number indicating the number of rows back from the current row to use for the calculation.

Example: Calculating the Increase Between the Current and a Prior Value of a Field

The following request uses the default value of SET PARTITION_ON (PENULTIMATE) to calculate the increase within the PRODUCT_CATEGORY sort field between the current row and the previous row.

```
SET PARTITION_ON=PENULTIMATE
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE INC = INCREASE(QUANTITY_SOLD,-1);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. The first value for INC is the value in the Accessories category for Quantity Sold, as there is no prior value. The second value for INC is the difference between the values for Headphones and Charger, the third is the difference between Universal Remote Controls and Headphones. Then, the calculations start over for Camcorder, which is the reset point.



Product Category	Product Subcategory	Quantity Sold	INC
Accessories	Charger	105,257	105,257.00
	Headphones	228,349	123,092.00
	Universal Remote Controls	178,061	-50,288.00
Camcorder	Handheld	250,167	250,167.00
	Professional	12,872	-237,295.00
	Standard	192,205	179,333.00
Computers	Smartphone	205,049	205,049.00
	Tablet	146,728	-58,321.00
Media Player	Blu Ray	679,495	679,495.00
	DVD Players	18,835	-660,660.00
	DVD Players - Portable	5,694	-13,141.00
	Streaming	67,910	62,216.00
Stereo Systems	Boom Box	9,370	9,370.00
	Home Theater Systems	399,092	389,722.00
	Receivers	150,568	-248,524.00
	Speaker Kits	244,199	93,631.00
	iPod Docking Station	311,103	66,904.00
Televisions	CRT TV	4,638	4,638.00
	Flat Panel TV	92,501	87,863.00
	Portable TV	8,049	-84,452.00
Video Production	Video Editing	199,749	199,749.00

PCT_INCREASE: Calculating the Percentage Difference Between the Current and a Prior Value of a Field

Given an aggregated input field and a negative offset, PCT_INCREASE calculates the percentage difference between the value in the current row of the report output and one or more prior rows, within a sort break or the entire table. The reset point for the calculation is determined by the value of the PARTITION_ON parameter described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.

The percentage increase is calculated using the following formula:

```
(current_value - prior_value) / prior_value
```

Note: The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

Syntax: How to Calculate the Percentage Difference Between the Current and a Prior Value of a Field

```
PCT_INCREASE([prefix.]field, offset)
```

where:

prefix

Is one of the following optional aggregation operators to apply to the field before using it in the calculation:

- **SUM.** which calculates the sum of the field values. SUM is the default value.
- **CNT.** which calculates a count of the field values.
- **AVE.** which calculates the average of the field values.
- **MIN.** which calculates the minimum of the field values.
- **MAX.** which calculates the maximum of the field values.
- **FST.** which retrieves the first value of the field.
- **LST.** which retrieves the last value of the field.

field

Numeric

The field to be used in the calculation.

offset

Numeric

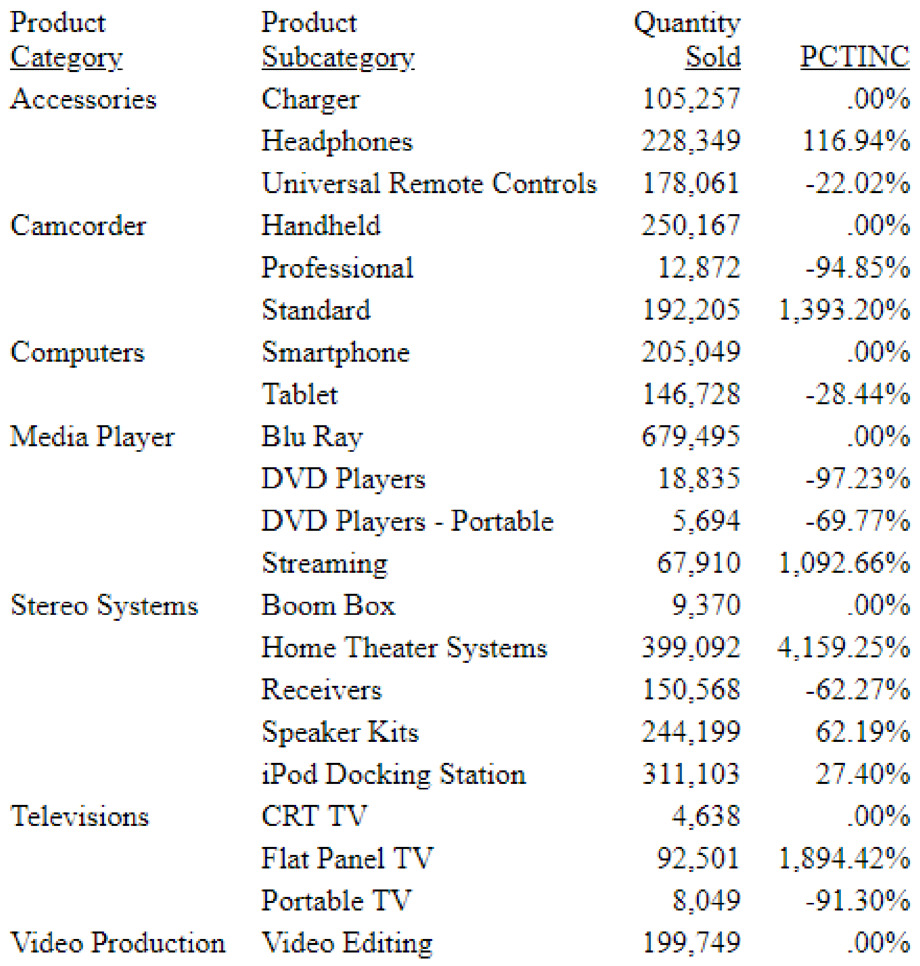
Is a negative number indicating the number of rows back from the current row to use for the calculation.

Example: PCT_INCREASE: Calculating the Percent Increase Between the Current and a Prior Value of a Field

The following request uses the default value of SET PARTITION_ON (PENULTIMATE) to calculate the percent increase within the PRODUCT_CATEGORY sort field between the current row and the previous row.

```
SET PARTITION_ON=PENULTIMATE
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE PCTINC/D8.2p = PCT_INCREASE(QUANTITY_SOLD,-1);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. The first value for PCTINC is zero percent, as there is no prior value. The second value for PCTINC is the percent difference between the values for Headphones and Charger, the third is the percent difference between Universal Remote Controls and Headphones. Then, the calculations start over for Camcorder, which is the reset point.



Product Category	Product Subcategory	Quantity Sold	PCTINC
Accessories	Charger	105,257	.00%
	Headphones	228,349	116.94%
	Universal Remote Controls	178,061	-22.02%
Camcorder	Handheld	250,167	.00%
	Professional	12,872	-94.85%
	Standard	192,205	1,393.20%
Computers	Smartphone	205,049	.00%
	Tablet	146,728	-28.44%
Media Player	Blu Ray	679,495	.00%
	DVD Players	18,835	-97.23%
	DVD Players - Portable	5,694	-69.77%
	Streaming	67,910	1,092.66%
Stereo Systems	Boom Box	9,370	.00%
	Home Theater Systems	399,092	4,159.25%
	Receivers	150,568	-62.27%
	Speaker Kits	244,199	62.19%
	iPod Docking Station	311,103	27.40%
Televisions	CRT TV	4,638	.00%
	Flat Panel TV	92,501	1,894.42%
	Portable TV	8,049	-91.30%
Video Production	Video Editing	199,749	.00%

PREVIOUS: Retrieving a Prior Value of a Field

Given an aggregated input field and a negative offset, PREVIOUS retrieves the value in a prior row, within a sort break or the entire table. The reset point for the calculation is determined by the value of the PARTITION_ON parameter described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.

Note: The values used in the retrieval depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

Syntax: How to Retrieve a Prior Value of a Field

```
PREVIOUS([prefix.]field, offset)
```

where:

prefix

Is one of the following optional aggregation operators to apply to the field before using it in the calculation:

- **SUM.** which calculates the sum of the field values. SUM is the default value.
- **CNT.** which calculates a count of the field values.
- **AVE.** which calculates the average of the field values.
- **MIN.** which calculates the minimum of the field values.
- **MAX.** which calculates the maximum of the field values.
- **FST.** which retrieves the first value of the field.
- **LST.** which retrieves the last value of the field.

field

Numeric or an alphanumeric field that contains all numeric digits.

The field to be used in the calculation.

offset

Numeric

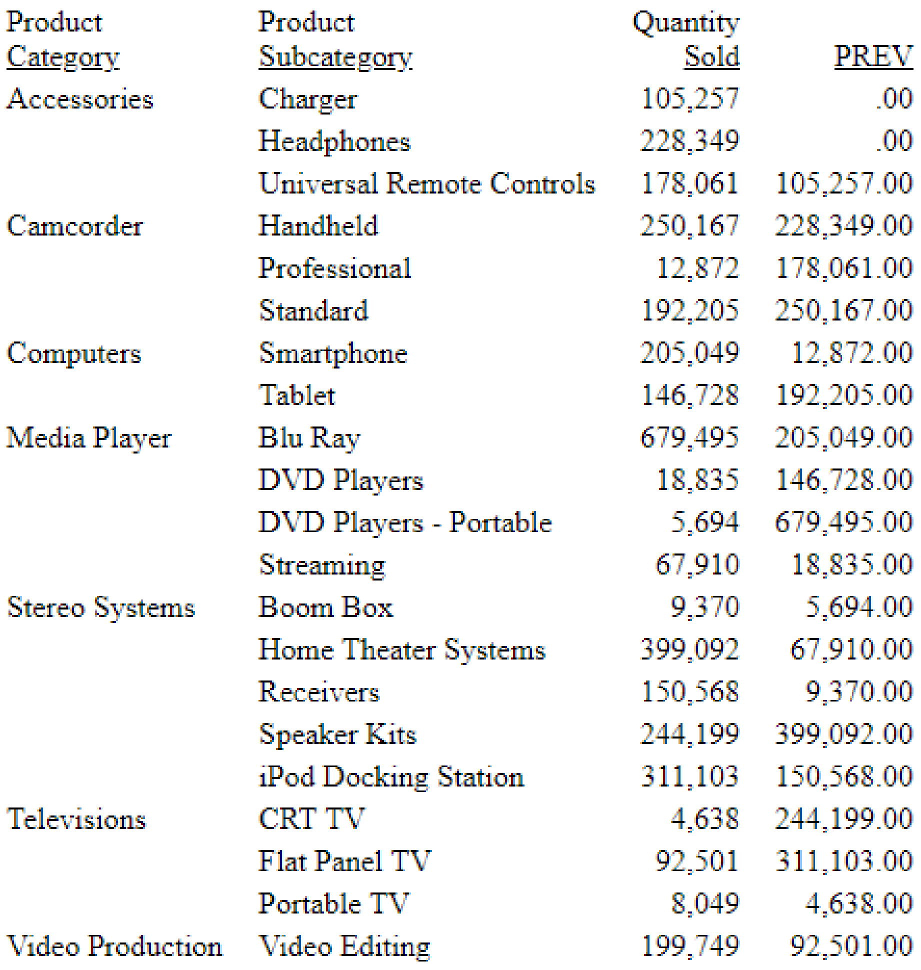
Is a negative number indicating the number of rows back from the current row to use for the retrieval.

Example: Retrieving a Prior Value of a Field

The following request sets the PARITITON_ON parameter to TABLE and retrieves the value of the QUANTITIY_SOLD field two rows back from the current row.

```
SET PARTITION_ON=TABLE
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE PREV = PREVIOUS(QUANTITY_SOLD,-2);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. The value of PREV in the first two rows is zero, as there are no prior rows for retrieval. From then on, each value of PREV is from the QUANTITY_SOLD value from two rows prior, with no reset points.



Product Category	Product Subcategory	Quantity Sold	PREV
Accessories	Charger	105,257	.00
	Headphones	228,349	.00
	Universal Remote Controls	178,061	105,257.00
Camcorder	Handheld	250,167	228,349.00
	Professional	12,872	178,061.00
	Standard	192,205	250,167.00
Computers	Smartphone	205,049	12,872.00
	Tablet	146,728	192,205.00
Media Player	Blu Ray	679,495	205,049.00
	DVD Players	18,835	146,728.00
	DVD Players - Portable	5,694	679,495.00
	Streaming	67,910	18,835.00
Stereo Systems	Boom Box	9,370	5,694.00
	Home Theater Systems	399,092	67,910.00
	Receivers	150,568	9,370.00
	Speaker Kits	244,199	399,092.00
	iPod Docking Station	311,103	150,568.00
Televisions	CRT TV	4,638	244,199.00
	Flat Panel TV	92,501	311,103.00
	Portable TV	8,049	4,638.00
Video Production	Video Editing	199,749	92,501.00

RUNNING_AVE: Calculating an Average Over a Group of Rows

Given an aggregated input field and a negative offset, RUNNING_AVE calculates the average of the values between the current row of the report output and one or more prior rows, within a sort break or the entire table. The reset point for the calculation is determined by the sort field specified, the entire table, or the value of the PARTITION_ON parameter described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.

Syntax: How to Calculate Running Average Between the Current and a Prior Value of a Field

RUNNING_AVE(*field*, *reset_key*, *lower*)

where:

field

Numeric

The field to be used in the calculation.

reset_key

Identifies the point at which the running average restarts. Valid values are:

- ❑ The name of a sort field in the request.
- ❑ PRESET, which uses the value of the PARTITION_ON parameter, as described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.
- ❑ TABLE, which indicates that there is no break on a sort field.

Note: The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

lower

Is the starting point in the partition for the running average. Valid values are:

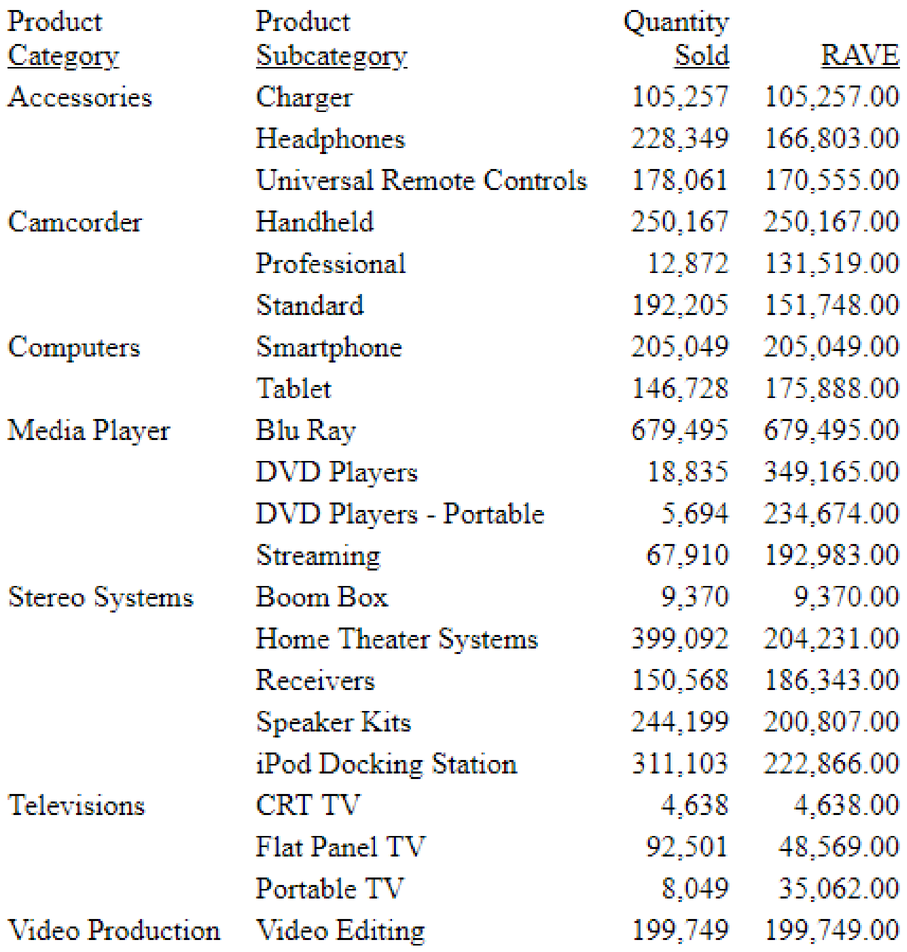
- ❑ A negative number, which identifies the offset from the current row.
- ❑ B, which specifies the beginning of the sort group.

Example: Calculating a Running Average

The following request calculates a running average of QUANTITY_SOLD within the PRODUCT_CATEGORY sort field, always starting from the beginning of the sort break.

```
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE RAVE = RUNNING_AVE(QUANTITY_SOLD,PRODUCT_CATEGORY,B);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. The first value for RAVE is the value in the Accessories category for Quantity Sold, as there is no prior value. The second value for RAVE is the average of the values for Headphones and Charger, the third is the average of the values for Headphones, Charger, and Universal Remote Controls. Then, the calculations start over for Camcorder, which is the reset point.



Product Category	Product Subcategory	Quantity Sold	RAVE
Accessories	Charger	105,257	105,257.00
	Headphones	228,349	166,803.00
	Universal Remote Controls	178,061	170,555.00
Camcorder	Handheld	250,167	250,167.00
	Professional	12,872	131,519.00
	Standard	192,205	151,748.00
Computers	Smartphone	205,049	205,049.00
	Tablet	146,728	175,888.00
Media Player	Blu Ray	679,495	679,495.00
	DVD Players	18,835	349,165.00
	DVD Players - Portable	5,694	234,674.00
	Streaming	67,910	192,983.00
Stereo Systems	Boom Box	9,370	9,370.00
	Home Theater Systems	399,092	204,231.00
	Receivers	150,568	186,343.00
	Speaker Kits	244,199	200,807.00
	iPod Docking Station	311,103	222,866.00
Televisions	CRT TV	4,638	4,638.00
	Flat Panel TV	92,501	48,569.00
	Portable TV	8,049	35,062.00
Video Production	Video Editing	199,749	199,749.00

RUNNING_MAX: Calculating a Maximum Over a Group of Rows

Given an aggregated input field and an offset, RUNNING_MAX calculates the maximum of the values between the current row of the report output and one or more prior rows, within a sort break or the entire table. The reset point for the calculation is determined by the sort field specified, the entire table, or the value of the PARTITION_ON parameter described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.

Syntax: How to Calculate Running Maximum Between the Current and a Prior Value of a Field

```
RUNNING_MAX(field, reset_key, lower)
```

where:

field

Numeric or an alphanumeric field that contains all numeric digits.

The field to be used in the calculation.

reset_key

Identifies the point at which the running maximum restarts. Valid values are:

- The name of a sort field in the request.
- PRESET, which uses the value of the PARTITION_ON parameter, as described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.
- TABLE, which indicates that there is no break on a sort field.

Note: The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

lower

Is the starting point in the partition for the running maximum. Valid values are:

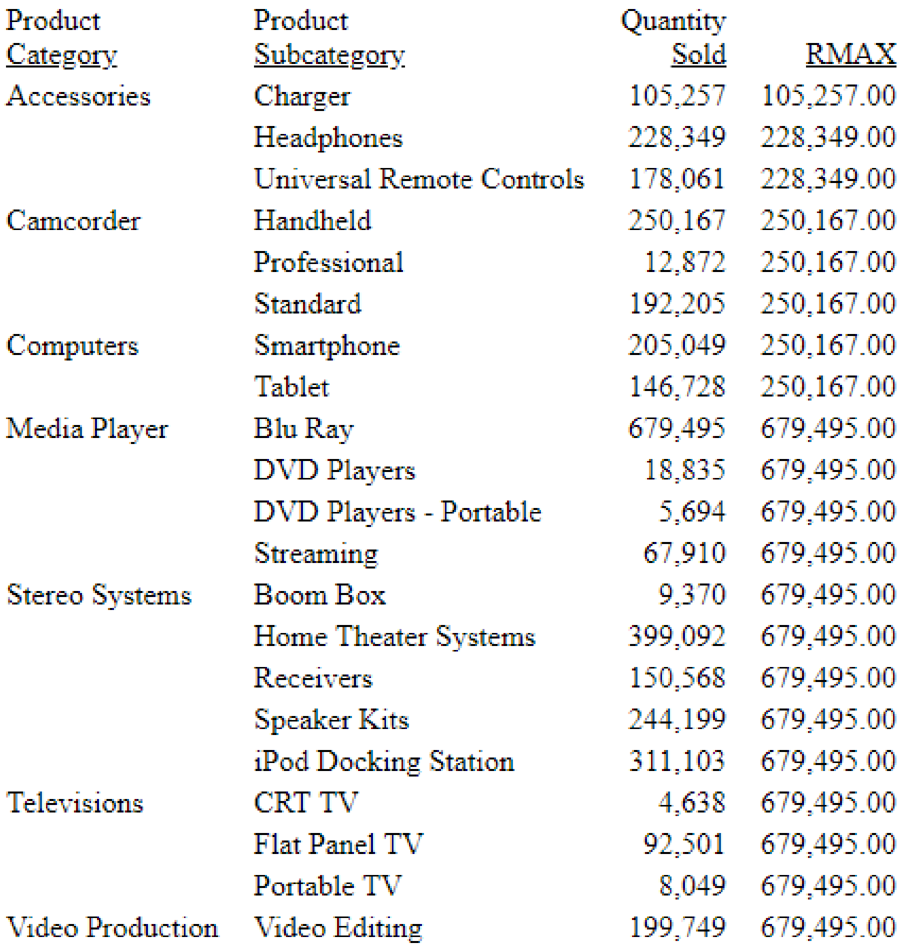
- A negative number, which identifies the offset from the current row.
- B, which specifies the beginning of the sort group.

Example: Calculating a Running Maximum

The following request calculates a running maximum for the rows from the beginning of the table to the current value of QUANTITY_SOLD, with no reset point.

```
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE RMAX = RUNNING_MAX(QUANTITY_SOLD,TABLE,B);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. The first value for RMAX is the value in the Accessories category for Quantity Sold, as there is no prior value. The second value for RMAX is the value for Headphones, as that value is larger. The third value for RMAX is still the value for Headphones, as that value is larger than the Quantity Sold value in the third row. Since the maximum value in the table occurs for Blu Ray, that value is repeated on all future rows, as there is no reset point.



```
Product            Product                     Quantity
Category           Subcategory                     Sold        RMAX
Accessories        Charger                      105,257  105,257.00
                   Headphones                   228,349  228,349.00
                   Universal Remote Controls    178,061  228,349.00
Camcorder          Handheld                     250,167  250,167.00
                   Professional                  12,872  250,167.00
                   Standard                     192,205  250,167.00
Computers          Smartphone                   205,049  250,167.00
                   Tablet                       146,728  250,167.00
Media Player       Blu Ray                      679,495  679,495.00
                   DVD Players                   18,835  679,495.00
                   DVD Players - Portable         5,694  679,495.00
                   Streaming                     67,910  679,495.00
Stereo Systems     Boom Box                       9,370  679,495.00
                   Home Theater Systems         399,092  679,495.00
                   Receivers                    150,568  679,495.00
                   Speaker Kits                 244,199  679,495.00
                   iPod Docking Station         311,103  679,495.00
Televisions        CRT TV                         4,638  679,495.00
                   Flat Panel TV                 92,501  679,495.00
                   Portable TV                    8,049  679,495.00
Video Production   Video Editing                199,749  679,495.00
```

RUNNING_MIN: Calculating a Minimum Over a Group of Rows

Given an aggregated input field and an offset, RUNNING_MIN calculates the minimum of the values between the current row of the report output and one or more prior rows, within a sort break or the entire table. The reset point for the calculation is determined by the sort field specified, the entire table, or the value of the PARTITION_ON parameter described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.

Syntax: How to Calculate Running Minimum Between the Current and a Prior Value of a Field

```
RUNNING_MIN(field, reset_key, lower)
```

where:

field

Numeric or an alphanumeric field that contains all numeric digits.

The field to be used in the calculation.

reset_key

Identifies the point at which the running minimum restarts. Valid values are:

- The name of a sort field in the request.
- PRESET, which uses the value of the PARTITION_ON parameter, as described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.
- TABLE, which indicates that there is no break on a sort field.

Note: The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

lower

Is the starting point in the partition for the running minimum. Valid values are:

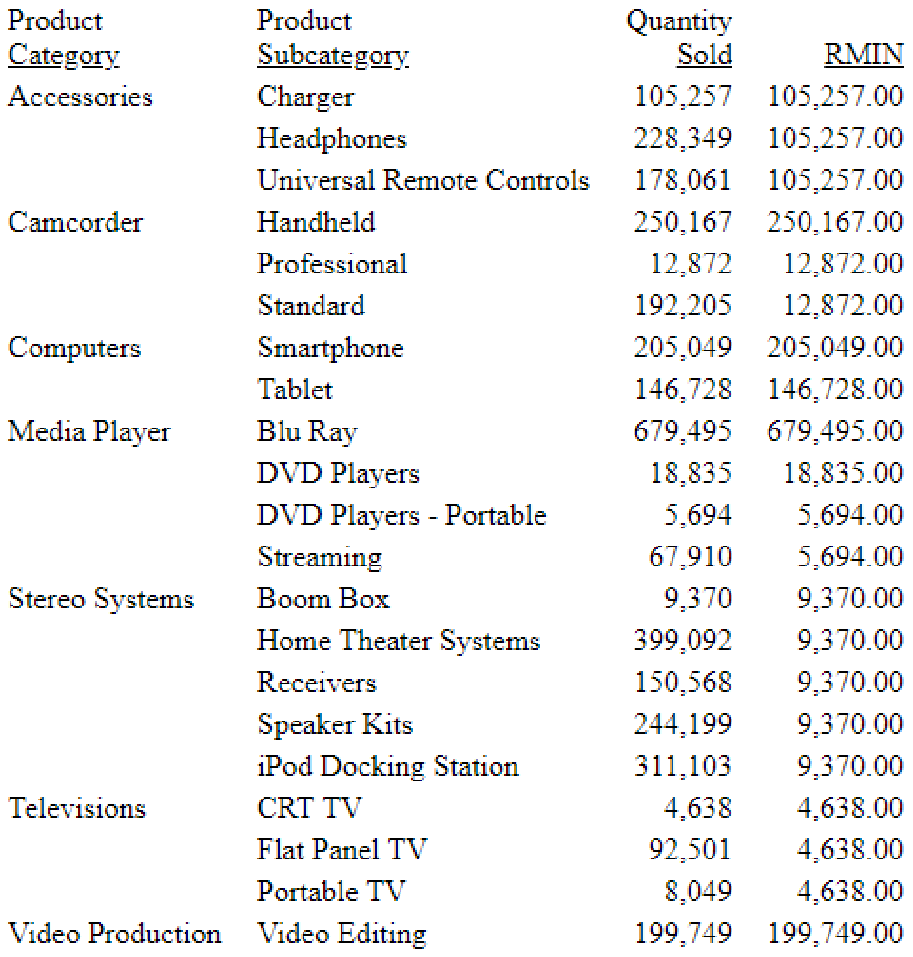
- A negative number, which identifies the offset from the current row.
- B, which specifies the beginning of the sort group.

Example: Calculating a Running Minimum

The following request calculates a running minimum of QUANTITY_SOLD within the PRODUCT_CATEGORY sort field (the sort break defined by SET PARTITION_ON = PENULTIMATE), always starting from the beginning of the sort break.

```
SET PARTITION_ON=PENULTIMATE
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE RMIN = RUNNING_MIN(QUANTITY_SOLD,PRESET,B);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. The first value for RMIN is the value in the Accessories category for Quantity Sold, as there is no prior value. The second value for RMIN is the value from the first row again (Charger), as that is smaller than the value in the second row. The third is the same again, as it is still the smallest. Then, the calculations start over for Camcorder, which is the reset point.



Product Category	Product Subcategory	Quantity Sold	RMIN
Accessories	Charger	105,257	105,257.00
	Headphones	228,349	105,257.00
	Universal Remote Controls	178,061	105,257.00
Camcorder	Handheld	250,167	250,167.00
	Professional	12,872	12,872.00
	Standard	192,205	12,872.00
Computers	Smartphone	205,049	205,049.00
	Tablet	146,728	146,728.00
Media Player	Blu Ray	679,495	679,495.00
	DVD Players	18,835	18,835.00
	DVD Players - Portable	5,694	5,694.00
	Streaming	67,910	5,694.00
Stereo Systems	Boom Box	9,370	9,370.00
	Home Theater Systems	399,092	9,370.00
	Receivers	150,568	9,370.00
	Speaker Kits	244,199	9,370.00
	iPod Docking Station	311,103	9,370.00
Televisions	CRT TV	4,638	4,638.00
	Flat Panel TV	92,501	4,638.00
	Portable TV	8,049	4,638.00
Video Production	Video Editing	199,749	199,749.00

RUNNING_SUM: Calculating a Sum Over a Group of Rows

Given an aggregated input field and an offset, RUNNING_SUM calculates the sum of the values between the current row of the report output and one or more prior rows, within a sort break or the entire table. The reset point for the calculation is determined by the sort field specified, the entire table, or the value of the PARTITION_ON parameter described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.

Syntax: How to Calculate Running Sum Between the Current and a Prior Value of a Field

```
RUNNING_SUM(field, reset_key, lower)
```

where:

field

Numeric

The field to be used in the calculation.

reset_key

Identifies the point at which the running sum restarts. Valid values are:

- The name of a sort field in the request.
- PRESET, which uses the value of the PARTITION_ON parameter, as described in *How to Specify the Partition Size for Simplified Statistical Functions* on page 331.
- TABLE, which indicates that there is no break on a sort field.

Note: The values used in the calculations depend on the sort sequence (ascending or descending) specified in the request. Be aware that displaying a date or time dimension in descending order may produce different results than those you may expect.

lower

Is the starting point in the partition for the running sum. Valid values are:

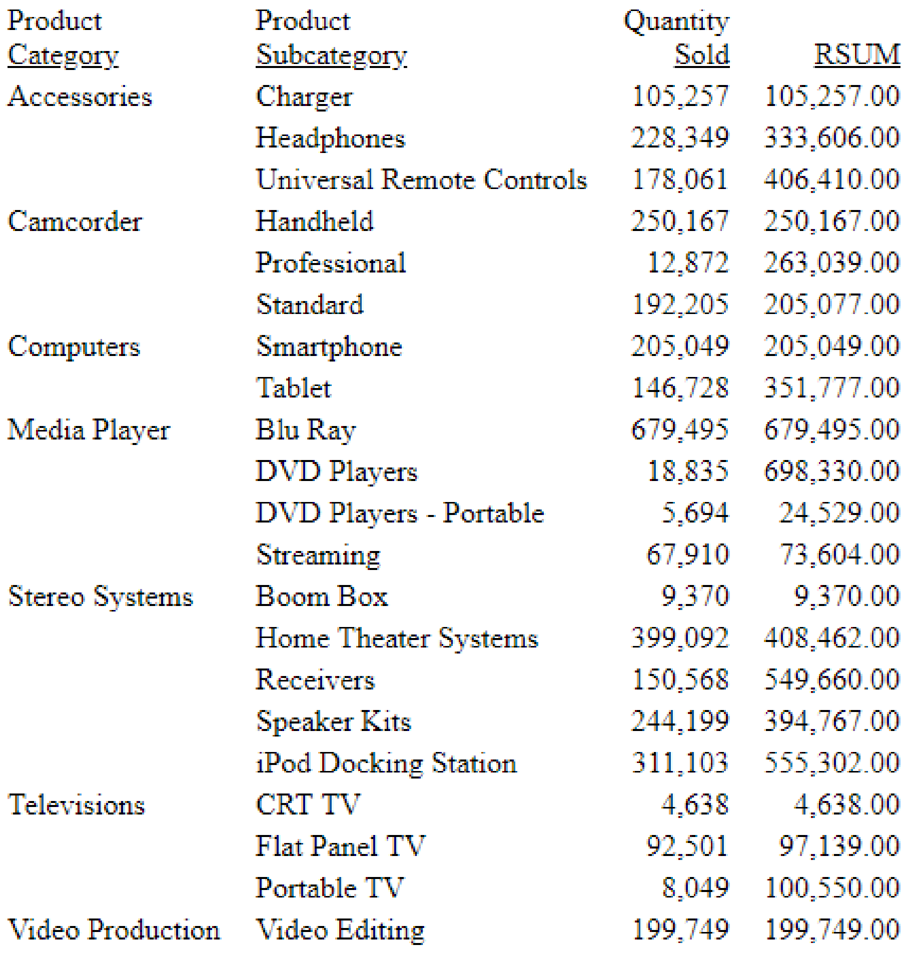
- A negative number, which identifies the offset from the current row.
- B, which specifies the beginning of the sort group.

Example: Calculating a Running Sum

The following request calculates a running sum of the current value and previous value of QUANTITY_SOLD within the reset point set by the PARTITION_ON parameter, which is the sort field PRODUCT_CATEGORY.

```
SET PARTITION_ON=PENULTIMATE
TABLE FILE wf_retail_lite
SUM QUANTITY_SOLD
COMPUTE RSUM = RUNNING_SUM(QUANTITY_SOLD,PRESET,-1);
BY PRODUCT_CATEGORY
BY PRODUCT_SUBCATEG
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. The first value for RSUM is the value in the Accessories category for Quantity Sold, as there is no prior value. The second value for RSUM is the sum of the values for Headphones and Charger, the third is the sum of the values for Headphones and Universal Remote Controls. Then, the calculations start over for Camcorder, which is the reset point.



```
Product           Product                        Quantity
Category          Subcategory                        Sold        RSUM
Accessories       Charger                         105,257  105,257.00
                  Headphones                      228,349  333,606.00
                  Universal Remote Controls       178,061  406,410.00
Camcorder         Handheld                        250,167  250,167.00
                  Professional                     12,872  263,039.00
                  Standard                        192,205  205,077.00
Computers         Smartphone                      205,049  205,049.00
                  Tablet                          146,728  351,777.00
Media Player      Blu Ray                         679,495  679,495.00
                  DVD Players                      18,835  698,330.00
                  DVD Players - Portable            5,694   24,529.00
                  Streaming                        67,910   73,604.00
Stereo Systems    Boom Box                          9,370    9,370.00
                  Home Theater Systems            399,092  408,462.00
                  Receivers                       150,568  549,660.00
                  Speaker Kits                    244,199  394,767.00
                  iPod Docking Station            311,103  555,302.00
Televisions       CRT TV                            4,638    4,638.00
                  Flat Panel TV                    92,501   97,139.00
                  Portable TV                       8,049  100,550.00
Video Production  Video Editing                   199,749  199,749.00
```

Chapter 3

Simplified Character Functions

Simplified character functions have streamlined parameter lists, similar to those used by SQL functions. In some cases, these simplified functions provide slightly different functionality than previous versions of similar functions.

The simplified functions do not have an output argument. Each function returns a value that has a specific data type.

When used in a request against a relational data source, these functions are optimized (passed to the RDBMS for processing).

In this chapter:

ASCII: Returning the ASCII Code for the Leftmost Character in a String

ASCII takes a character string and returns the ASCII code in integer format for the leftmost character in the string.

Syntax: **How to Return the ASCII Code for the Leftmost Character in a String**

```
ASCII(charexp)
```

where:

charexp

Is any character string.

Example: **Returning the ASCII Code for the Leftmost Character in a String**

ASCII returns the ASCII code of the leftmost character of CATEGORY.

```
ASCII(CATEGORY)
```

For Coffee, the result is 67.

CHAR_LENGTH: Returning the Length in Characters of a String

The CHAR_LENGTH function returns the length, in characters, of a string. In Unicode environments, this function uses character semantics, so that the length in characters may not be the same as the length in bytes. If the string includes trailing blanks, these are counted in the returned length. Therefore, if the format source string is type A*n*, the returned value will always be *n*.

Syntax: How to Return the Length of a String in Characters

```
CHAR_LENGTH(string)
```

where:

`string`

Alphanumeric

Is the string whose length is returned.

The data type of the returned length value is Integer.

Example: Returning the Length of a String

LASTNAME has format A15V and contains the last name with trailing blanks removed. CHAR_LENGTH returns the number of characters:

```
CHAR_LENGTH(LASTNAME)
```

For SMITH, the result is 5.

CONCAT: Concatenating Strings

CONCAT concatenates two strings. The output is returned as variable length alphanumeric.

Syntax: How to Concatenate Strings

```
CONCAT(string1, string2)
```

where:

`string2`

Alphanumeric

Is a string to be concatenated.

string1

Alphanumeric

Is a string to be concatenated.

Example: **Concatenating Strings**

CONCAT concatenates CITY and STATE.

```
CONCAT(CITY,STATE)
```

For Montgomery Alabama, the result is Montgomery Alabama.

DIFFERENCE: Measuring the Phonetic Similarity Between Character Strings

DIFFERENCE returns an integer value measuring the difference between the SOUNDEX or METAPHONE values of two character expressions.

Syntax: **How to Measure the Phonetic Similarity Between Character String**

```
DIFFERENCE(chrexp1, chrexp2)
```

where:

chrexp1, chrexp2

Alphanumeric

Are the character strings to be compared.

Zero (0) represents the least similarity. For SOUNDEX, 4 represents the most similarity, and for METAPHONE, 16 represents the most similarity.

The use of SOUNDEX or METAPHONE depends on the PHONETIC_ALGORITHM setting. METAPHONE is the default algorithm.

Example: **Measuring the Phonetic Similarity Between Character Strings**

DIFFERENCE compares the character strings *Green* and *Greene*.

```
DIFFERENCE('Green','Greene')
```

For the phonetic algorithm METAPHONE (the default), the result is 16.

DIGITS: Converting a Number to a Character String

Given a number, DIGITS converts it to a character string of the specified length. The format of the field that contains the number must be Integer.

Syntax: How to Convert a Number to a Character String

```
DIGITS(number, length)
```

where:

number

Integer

Is the number to be converted, stored in a field with data type Integer.

length

Integer between 1 and 10

Is the length of the returned character string. If *length* is longer than the number of digits in the number being converted, the returned value is padded on the left with zeros. If *length* is shorter than the number of digits in the number being converted, the returned value is truncated on the left.

Example: Converting a Number to a Character String

DIGITS converts the integer expression ID_PRODUCT+1 to a six-character string:

```
DIGITS(ID_PRODUCT,6)
```

For the number 1106, the result is the character string '001106'.

Reference: Usage Notes for DIGITS

- ❏ Only I format numbers will be converted. D, P, and F formats generate error messages and should be converted to I before using the DIGITS function. The limit for the number that can be converted is 2 GB.
- ❏ Negative integers are turned into positive integers.
- ❏ Integer formats with decimal places are truncated.
- ❏ DIGITS is not supported in Dialogue Manager.

GET_TOKEN: Extracting a Token Based on a String of Delimiters

GET_TOKEN extracts a token (substring) based on a string that can contain multiple characters, each of which represents a single-character delimiter.

Syntax: How to Extract a Token Based on a String of Delimiters

```
GET_TOKEN(string, delimiter_string, occurrence)
```

where:

string

Alphanumeric

Is the input string from which the token will be extracted. This can be an alphanumeric field or constant.

delimiter_string

Alphanumeric constant

Is a string that contains the list of delimiter characters. For example, '; ,' contains three delimiter characters, semi-colon, blank space, and comma.

occurrence

Integer constant

Is a positive integer that specifies the token to be extracted. A negative integer will be accepted in the syntax, but will not extract a token. The value zero (0) is not supported.

Example: Extracting a Token Based on a String of Delimiters

GET_TOKEN extracts a token based on a string of delimiters.

```
GET_TOKEN(InputString, ',;/', 4)
```

For input string 'ABC,DEF;GHI/JKL', the result is JKL.

INITCAP: Capitalizing the First Letter of Each Word in a String

INITCAP capitalizes the first letter of each word in an input string and makes all other letters lowercase. A word starts at the beginning of the string, after a blank space, or after a special character.

Syntax: How to Capitalize the First Letter of Each Word in a String

```
INITCAP(input_string)
```

where:

input_string

Alphanumeric

Is the string to capitalize.

Example: Capitalizing the First Letter of Each Word in a String

INITCAP capitalizes the first letter of each word.

```
INITCAP(NewName)
```

For the string abc,def!ghi'jKL MNO, the result is Abc,Def!Ghi'Jkl Mno.

For MCKNIGHT, the result is Mcknight.

LAST_NONBLANK: Retrieving the Last Field Value That is Neither Blank nor Missing

LAST_NONBLANK retrieves the last field value that is neither blank nor missing. If all previous values are either blank or missing, LAST_NONBLANK returns a missing value.

Syntax: How to Return the Last Value That is Neither Blank nor Missing

```
LAST_NONBLANK(field)
```

where:

field

Is the field name whose last non-blank value is to be retrieved. If the current value is not blank or missing, the current value is returned.

Note: LAST_NONBLANK cannot be used in a compound expression, for example, as part of an IF condition.

Example: Retrieving the Last Non-Blank Value

Consider the following delimited file named input1.csv that has two fields named FIELD_1 and FIELD_2.

```
,
A,
,
 ,
B,
C,
```

The input1 Master File follows.

```
FILENAME=INPUT1, SUFFIX=DFIX    ,
 DATASET=baseapp/input1.csv(LRECL 15 RECFM V, BV_NAMESPACE=OFF, $
  SEGMENT=INPUT1, SEGTYPE=S0, $
    FIELDNAME=FIELD_1, ALIAS=E01, USAGE=A1V, ACTUAL=A1V,
      MISSING=ON, $
    FIELDNAME=FIELD_2, ALIAS=E02, USAGE=A1V, ACTUAL=A1V,
      MISSING=ON, $
```

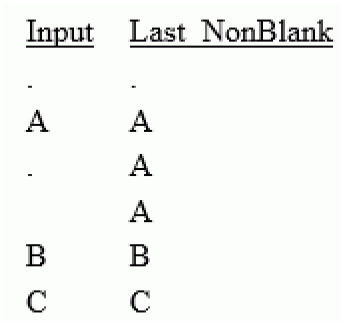
The input1 Access File follows.

```
SEGNAME=INPUT1,
  DELIMITER=',',
  HEADER=NO,
  PRESERVESPACE=NO,
  CDN=COMMAS_DOT,
  CONNECTION=<local>, $
```

The following request displays the FIELD_1 values and computes the last non-blank value for each FIELD_1 value.

```
TABLE FILE baseapp/INPUT1
PRINT FIELD_1 AS Input
COMPUTE
Last_NonBlank/A1 MISSING ON = LAST_NONBLANK(FIELD_1);
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image.



```
Input   Last_NonBlank

.       .
A       A
.       A
        A
B       B
C       C
```

LEFT: Returning Characters From the Left of a Character String

Given a source character string, or an expression that can be converted to varchar (variable-length alphanumeric), and an integer number, LEFT returns that number of characters from the left end of the string.

Syntax: How to Return Characters From the Left of a Character String

```
LEFT(chr_exp, int_exp)
```

where:

chr_exp

Alphanumeric or an expression that can be converted to variable-length alphanumeric.

Is the source character string.

int_exp

Integer

Is the number of characters to be returned.

Example: **Returning Characters From the Left of a Character String**

LEFT returns the two leftmost characters from SOURCE:

```
LEFT(SOURCE,2)
```

For 'abcdefg', the result is *ab*.

LOWER: Returning a String With All Letters Lowercase

The LOWER function takes a source string and returns a string of the same data type with all letters translated to lowercase.

Syntax: **How to Return a String With All Letters Lowercase**

```
LOWER(string)
```

where:

string

Alphanumeric

Is the string to convert to lowercase.

The returned string is the same data type and length as the source string.

Example: **Converting a String to Lowercase**

LOWER converts LAST_NAME to lowercase.

```
LOWER(LAST_NAME)
```

For STEVENS, the result is stevens.

LPAD: Left-Padding a Character String

LPAD uses a specified character and output length to return a character string padded on the left with that character.

Syntax: **How to Pad a Character String on the Left**

```
LPAD(string, out_length, pad_character)
```

where:

`string`

Fixed length alphanumeric

Is a string to pad on the left side.

`out_length`

Integer

Is the length of the output string after padding.

`pad_character`

Fixed length alphanumeric

Is a single character to use for padding.

Example: **Left-Padding a String**

LPAD left-pads the PRODUCT_CATEGORY column with @ symbols:

```
LPAD(PRODUCT_CATEGORY,25,'@')
```

For *Stereo Systems*, the output is @@@@@@@@@@@*Stereo Systems*.

Reference: **Usage Notes for LPAD**

- To use the single quotation mark (') as the padding character, you must double it and enclose the two single quotation marks within single quotation marks (LPAD(COUNTRY, 20,'''')). You can use an amper variable in quotation marks for this parameter, but you cannot use a field, virtual or real.
- Input can be fixed or variable length alphanumeric.
- Output, when optimized to SQL, will always be data type VARCHAR.
- If the output is specified as shorter than the original input, the original data will be truncated, leaving only the padding characters. The output length can be specified as a positive integer or an unquoted &variable (indicating a numeric).

LTRIM: Removing Blanks From the Left End of a String

The LTRIM function removes all blanks from the left end of a string.

Syntax: How to Remove Blanks From the Left End of a String

```
LTRIM(string)
```

where:

string

Alphanumeric

Is the string to trim on the left.

The data type of the returned string is AnV, with the same maximum length as the source string.

Example: Removing Blanks From the Left End of a String

RDIRECTOR has the director name right justified. LTRIM removes the leading blanks.

```
LTRIM(RDIRECTOR)
```

For   BROOKS R. the result is BROOKS R.

OVERLAY: Replacing Characters in a String

Given a starting position, length, source string, and insertion string, OVERLAY replaces the number of characters defined by *length* in the source string with the insertion string, starting from the starting position.

Syntax: How to Replace Characters in a String

```
OVERLAY(src, ins, start, len)
```

where:

src

Alphanumeric

Is the source string whose characters will be replaced.

ins

Alphanumeric

Is the insertion string with the replacement characters.

start

Numeric

Is the starting position for the replacement in the source string.

len

Numeric

Is the number of characters to replace in the source string with the entire insertion string.

Example: Replacing Characters in a String

OVERLAY replaces the first three characters in 'ENGLAND' with the characters 'SCOT'.

```
OVERLAY('ENGLAND', 'SCOT', 1, 3)
```

The result is 'SCOTLAND'.

PATTERNS: Returning a Pattern That Represents the Structure of the Input String

PATTERNS returns a string that represents the structure of the input argument. The returned pattern includes the following characters:

- **A** is returned for any position in the input string that has an uppercase letter.
- **a** is returned for any position in the input string that has a lowercase letter.
- **9** is returned for any position in the input string that has a digit.

Note that special characters (for example, +-/=%) are returned exactly as they were in the input string.

The output is returned as variable length alphanumeric.

Syntax: How to Return a String That Represents the Pattern Profile of the Input Argument

```
PATTERNS(string)
```

where:

string

Alphanumeric

Is a string whose pattern will be returned.

Example: Returning a Pattern Representing an Input String

PATTERNS returns the pattern representing the field ADDRESS_LINE_1.

```
PATTERNS(ADDRESS_LINE_1)
```

For 1010 Milam St # Ifp-2352

The result is 9999 Aaaaa Aa # Aaa-9999.

POSITION: Returning the First Position of a Substring in a Source String

The POSITION function returns the first position (in characters) of a substring in a source string.

Syntax: How to Return the First Position of a Substring in a Source String

```
POSITION(pattern, string)
```

where:

pattern

Alphanumeric

Is the substring whose position you want to locate. The string can be as short as a single character, including a single blank.

string

Alphanumeric

Is the string in which to find the pattern.

The data type of the returned value is Integer.

Example: Returning the First Position of a Substring

POSITION determines the position of the first capital letter I in LAST_NAME.

```
POSITION('I', LAST_NAME)
```

For STEVENS, the result is 0.

For SMITH, the result is 3.

POSITION: Returning the Position of a Search String in a Source String

Given a search string, a source string, and a starting position, POSITION returns the position of the search string within the source string. The search starts at the given starting position and searches from left to right. If the string is not found, POSITION returns zero (0). The search is case sensitive.

Syntax: How to Return the Position of a Search String in a Source String

```
POSITION(search, source, start)
```

where:

search
: Alphanumeric

  Is the search string.

source
: Alphanumeric

  Is the source string.

start
: Numeric

  Is the starting position in the source string for the search.

Example: Returning the Position of a Search String in a Source String

POSITION finds the first instance of the uppercase letter A in CustomerName after position 3.

```
POSITION('A', CustomerName, 3)
```

For *Sandra Arzola*, the result is 8.

REGEX: Matching a String to a Regular Expression

The REGEX function matches a string to a regular expression and returns true (1) if it matches and false (0) if it does not match.

A regular expression is a sequence of special characters and literal characters that you can combine to form a search pattern.

Many references for regular expressions exist on the web.

Syntax: How to Match a String to a Regular Expression

```
REGEX(string, regular_expression)
```

where:

string
: Alphanumeric

  Is the character string to match.

regular_expression

Alphanumeric

Is a regular expression, enclosed in single quotation marks, constructed using literals and meta-characters. The following meta-characters are supported

- . represents any single character
- * represents zero or more occurrences
- + represents one or more occurrences
- ? represents zero or one occurrence
- ^ represents beginning of line
- $ represents end of line
- [] represents any one character in the set listed within the brackets
- [^] represents any one character not in the set listed within the brackets
- | represents the Or operator
- \ is the Escape Special Character
- () contains a character sequence

For example, the regular expression '^Ste(v|ph)en$' matches values starting with *Ste* followed by either *ph* or *v*, and ending with *en*.

Note: The output value is numeric.

Example: Matching a String Against a Regular Expression

REGEX matches the FIRSTNAME field against the regular expression '^Sara(h?)$', which matches Sara or Sarah:

```
REGEX(FIRSTNAME,'^Sara(h?)$')
```

For Sara, the result is 1.

For Amber, the result is 0.

REGEXP_COUNT: Counting the Number of Matches to a Pattern in a String

REGEXP_COUNT returns the integer count of matches to a specified regular expression pattern within a source string.

Syntax: How to Count the Number of Matches to a Pattern in a String

```
REGEXP_COUNT(string, pattern)
```

where:

string

Alphanumeric

Is the input string to be searched.

pattern

Alphanumeric

Is a regular expression, enclosed in single quotation marks, constructed using literals and meta-characters. The following meta-characters are supported

- . represents any single character
- * represents zero or more occurrences
- + represents one or more occurrences
- ? represents zero or one occurrence
- ^ represents beginning of line
- $ represents end of line
- [] represents any one character in the set listed within the brackets
- [^] represents any one character not in the set listed within the brackets
- | represents the Or operator
- \ is the Escape Special Character
- () contains a character sequence

Example: Counting the Number of Matches to a Pattern in a String

The following examples use the following Regular Expression symbols.

- $, which searches for a specified expression that occurs at the end of a string.
- ^, which searches for a specified expression that occurs at the beginning of a string.

REGEXP_COUNT counts the number of occurrences of the characters 'umpty' that occur at the end of the string 'Humpty Dumpty'.

```
REGEXP_COUNT('Humpty Dumpty', 'umpty$')
```

The result is 1.

REGEXP_COUNT counts the number of occurrences of the characters 'umpty' that occur at the beginning of the string 'Humpty Dumpty'.

```
REGEXP_COUNT('Humpty Dumpty', '^umpty')
```

The result is 0.

REGEXP_INSTR: Returning the First Position of a Pattern in a String

REGEXP_INSTR returns the integer position of the first match to a specified regular expression pattern within a source string. The first character position in a string is indicated by the value 1. If there is no match within the source string, the value 0 is returned.

Syntax: How to Return the Position of a Pattern in a String

```
REGEXP_INSTR(string, pattern)
```

where:

string

Alphanumeric

Is the input string to be searched.

pattern

Alphanumeric

Is a regular expression, enclosed in single quotation marks, constructed using literals and meta-characters. The following meta-characters are supported

- . represents any single character
- * represents zero or more occurrences
- + represents one or more occurrences
- ? represents zero or one occurrence
- ^ represents beginning of line
- $ represents end of line

- [] represents any one character in the set listed within the brackets
- [^] represents any one character not in the set listed within the brackets
- | represents the Or operator
- \ is the Escape Special Character
- () contains a character sequence

Example: Finding the Position of a Pattern in a String

The following examples use the following Regular Expression symbols.

- $, which searches for a specified expression that occurs at the end of a string.
- ^, which searches for a specified expression that occurs at the beginning of a string.

REGEXP_INSTR finds the position of the characters 'umpty' that occur at the end of the string 'Humpty Dumpty'.

```
REGEXP_INSTR('Humpty Dumpty', 'umpty$')
```

The result is 9.

REGEXP_INSTR finds the position of the characters 'umpty' that occur at the beginning of the string 'Humpty Dumpty'.

```
REGEXP_INSTR('Humpty Dumpty', '^umpty')
```

The result is 0.

REGEXP_REPLACE: Replacing All Matches to a Pattern in a String

REGEXP_REPLACE returns a string generated by replacing all matches to a regular expression pattern in the source string with the given replacement string. The replacement string can be a null string.

Syntax: How to Replace Matches to a Pattern in a String

```
REGEXP_REPLACE(string, pattern, replacement)
```

where:

string

Alphanumeric

Is the input string to be searched.

pattern

Alphanumeric

Is a regular expression, enclosed in single quotation marks, constructed using literals and meta-characters. The following meta-characters are supported

- . represents any single character
- * represents zero or more occurrences
- + represents one or more occurrences
- ? represents zero or one occurrence
- ^ represents beginning of line
- $ represents end of line
- [] represents any one character in the set listed within the brackets
- [^] represents any one character not in the set listed within the brackets
- | represents the Or operator
- \ is the Escape Special Character
- () contains a character sequence

replacement

Alphanumeric

Is the replacement string.

Example: **Replacing Matches to a Pattern in a String**

The following example uses the following Regular Expression symbol.

- ^, which searches for a specified expression that occurs at the beginning of a string.

REGEXP_REPLACE replaces the characters 'ENG' at the beginning of the field COUNTRY with the replacement string 'SCOT'.

```
REGEXP_REPLACE(COUNTRY, '^ENG', 'SCOT')
```

For 'ENGLAND', the result is 'SCOTLAND'.

REGEXP_SUBSTR: Returning the First Match to a Pattern in a String

REGEXP_SUBSTR returns a string that contains the first match to a specified regular expression pattern within a source string. If there is no match within the source string, a null string is returned.

Syntax: **How to Returning the First Match to a Pattern in a String**

```
REGEXP_SUBSTR(string, pattern)
```

where:

string

Alphanumeric

Is the input string to be searched.

pattern

Alphanumeric

Is a regular expression, enclosed in single quotation marks, constructed using literals and meta-characters. The following meta-characters are supported

- . represents any single character
- * represents zero or more occurrences
- + represents one or more occurrences
- ? represents zero or one occurrence
- ^ represents beginning of line
- $ represents end of line
- [] represents any one character in the set listed within the brackets
- [^] represents any one character not in the set listed within the brackets
- | represents the Or operator
- \ is the Escape Special Character
- () contains a character sequence

Example: Returning the First Match of a Pattern in a String

The following example uses the following Regular Expression symbols.

- [A-Z], which matches any uppercase letter.
- $, which searches for a specified expression that occurs at the end of a string.

REGEXP_SUBSTR searches for a string with any uppercase letter followed by the characters 'umpty' at the end of the string 'Humpty Dumpty'.

```
REGEXP_SUBSTR('Humpty Dumpty', '[A-Z]umpty$')
```

The result is 'Dumpty'.

REPEAT: Repeating a String a Given Number of Times

Given a source string and an integer number, REPEAT returns a string with the source string repeated that number of times. The string containing the repeated strings must be large enough to fit the repetitions or it will contain a truncated value.

Syntax: How to Repeat a Character String a Given Number of Times

```
REPEAT(source_str, number)
```

where:

source_str

Alphanumeric

Is the source string to be repeated. If *source_str* is a field, the entire field, including blanks, will be repeated.

number

Numeric

Is the number of times to repeat the source string.

Example: Repeating a String a Given Number of Times

REPEAT returns a string with FIRST_NAME repeated three times.

```
REPEAT(FIRST_NAME, 3)
```

For MARY, the result is *MARY MARY MARY*.

REPLACE: Replacing a String

REPLACE replaces all instances of a search string in an input string with the given replacement string. The output is always variable length alphanumeric with a length determined by the input parameters.

Syntax: **How to Replace all Instances of a String**

```
REPLACE(input_string , search_string , replacement)
```

where:

input_string

Alphanumeric or text (An, AnV, TX)

Is the input string.

search_string

Alphanumeric or text (An, AnV, TX)

Is the string to search for within the input string.

replacement

Alphanumeric or text (An, AnV, TX)

Is the replacement string to be substituted for the search string. It can be a null string ('').

Example: **Replacing a String**

REPLACE replaces the string 'South' in the Country Name with the string 'S.'

```
REPLACE(COUNTRY_NAME, 'SOUTH', 'S.');
```

For South Africa, the result is S. Africa.

Example: **Replacing All Instances of a String**

REPLACE removes the characters 'DAY' from the string DAY1:

```
REPLACE(DAY1, 'DAY', '' )
```

For 'SUNDAY MONDAY TUESDAY', the result is 'SUN MON TUES'.

RIGHT: Returning Characters From the Right of a Character String

Given a source character string, or an expression that can be converted to varchar (variable-length alphanumeric), and an integer number, RIGHT returns that number of characters from the right end of the string.

Syntax: **How to Return Characters From the Right of a Character String**

`RIGHT(`*`chr_exp`*`, `*`int_exp`*`)`

where:

`chr_exp`

Alphanumeric or an expression that can be converted to variable-length alphanumeric.

Is the source character string.

`int_exp`

Integer

Is the number of characters to be returned.

Example: **Returning Characters From the Right of a Character String**

RIGHT returns the two rightmost characters from SOURCE:

```
RIGHT(SOURCE,2)
```

For 'abcdefg', the result is *fg*.

RPAD: Right-Padding a Character String

RPAD uses a specified character and output length to return a character string padded on the right with that character.

Syntax: **How to Pad a Character String on the Right**

`RPAD(`*`string`*`, `*`out_length`*`, `*`pad_character`*`)`

where:

`string`

Alphanumeric

Is a string to pad on the right side.

`out_length`

Integer

Is the length of the output string after padding.

`pad_character`

Alphanumeric

Is a single character to use for padding.

Example: Right-Padding a String

RPAD right-pads the PRODUCT_CATEGORY column with @ symbols:

```
RPAD(PRODUCT_CATEGORY,25,'@')
```

For *Stereo Systems*, the output is *Stereo Systems@@@@@@@@@@@*.

Reference: Usage Notes for RPAD

- The input string can be data type AnV, VARCHAR, TX, and An.
- Output can only be AnV or An.
- When working with relational VARCHAR columns, there is no need to trim trailing spaces from the field if they are not desired. However, with An and AnV fields derived from An fields, the trailing spaces are part of the data and will be included in the output, with the padding being placed to the right of these positions. You can use TRIM or TRIMV to remove these trailing spaces prior to applying the RPAD function.

RTRIM: Removing Blanks From the Right End of a String

The RTRIM function removes all blanks from the right end of a string.

Syntax: How to Remove Blanks From the Right End of a String

```
RTRIM(string)
```

where:

string

Alphanumeric

Is the string to trim on the right.

The data type of the returned string is AnV, with the same maximum length as the source string.

Example: Removing Blanks From the Right End of a String

RTRIM removes trailing blanks from DIRECTOR.

```
RTRIM(DIRECTOR)
```

For BROOKS R.   , the result is BROOKS R.

SPACE: Returning a String With a Given Number of Spaces

Given an integer count, SPACE returns a string consisting of that number of spaces.

Note: To retain the spaces in HTML report output, the SHOWBLANKS parameter must be set to ON.

Syntax: How to Return a String With a Given Number of Spaces

```
SPACE(count)
```

where:

count

Numeric

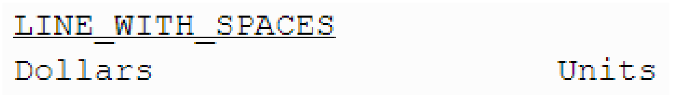
Is the number of spaces to return.

Example: Returning a String With a Given Number of Spaces

SPACE adds 20 blank spaces between the words 'Dollars' and 'Units' using the monospaced Courier font.

```
SET SHOWBLANKS = ON
SQL
SELECT
('Dollars' || SPACE(20) || 'Units') AS LINE_WITH_SPACES ;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF, FONT=COURIER,$
ENDSTYLE
END
```

The output is shown in the following image.



```
LINE_WITH_SPACES
Dollars                    Units
```

SPLIT: Extracting an Element From a String

The SPLIT function returns a specific type of element from a string. The output is returned as variable length alphanumeric.

Syntax: How to Extract an Element From a String

```
SPLIT(element, string)
```

where:

element

Can be one of the following keywords:

- **EMAIL_DOMAIN.** Is the domain name portion of an email address in the string.
- **EMAIL_USERID.** Is the user ID portion of an email address in the string.
- **URL_PROTOCOL.** Is the URL protocol for a URL in the string.
- **URL_HOST.** Is the host name of the URL in the string.
- **URL_PORT.** Is the port number of the URL in the string.
- **URL_PATH.** Is the URL path for a URL in the string.
- **NAME_FIRST.** Is the first token (group of characters) in the string. Tokens are delimited by blanks.
- **NAME_LAST.** Is the last token (group of characters) in the string. Tokens are delimited by blanks.

string

Alphanumeric

Is the string from which the element will be extracted.

Example: Extracting an Element From a String

SPLIT extracts the URL protocol from the string STRING1.

```
SPLIT(URL_PROTOCOL, STRING1)
```

For the URL *'http://www.informationbuilders.com'* in STRING1, the result is *http*.

SUBSTRING: Extracting a Substring From a Source String

The SUBSTRING function extracts a substring from a source string. If the ending position you specify for the substring is past the end of the source string, the position of the last character of the source string becomes the ending position of the substring.

Syntax: **How to Extract a Substring From a Source String**

```
SUBSTRING(string, position, length)
```

where:

string

Alphanumeric

Is the string from which to extract the substring. It can be a field, a literal in single quotation marks ('), or a variable.

position

Positive Integer

Is the starting position of the substring in *string*.

length

Integer

Is the limit for the length of the substring. The ending position of the substring is calculated as *position* + *length* - 1. If the calculated position beyond the end of the source string, the position of the last character of *string* becomes the ending position.

The data type of the returned substring is AnV.

Example: **Extracting a Substring From a Source String**

POSITION determines the position of the first letter I in LAST_NAME.

```
SUBSTRING(LAST_NAME, I_IN_NAME, I_IN_NAME+2)
```

For BANNING, the result is 5.

TOKEN: Extracting a Token From a String

The token function extracts a token (substring) from a string of characters. The tokens are separated by a delimiter consisting of one or more characters and specified by a token number reflecting the position of the token in the string.

Syntax: **How to Extract a Token From a String**

```
TOKEN(string, delimiter, number)
```

where:

`string`

Fixed length alphanumeric

Is the character string from which to extract the token.

`delimiter`

Fixed length alphanumeric

Is a delimiter consisting of one or more characters.

TOKEN can be optimized if the delimiter consists of a single character.

`number`

Integer

Is the token number to extract.

Example: Extracting a Token From a String

TOKEN extracts the second token from the PRODUCT_SUBCATEG column, where the delimiter is a blank:

```
TOKEN(PRODUCT_SUBCATEG,' ',2)
```

For *iPod Docking Station*, the result is *Docking*.

TRIM_: Removing a Leading Character, Trailing Character, or Both From a String

The TRIM_ function removes all occurrences of a single character from either the beginning or end of a string, or both.

Note:

- Leading and trailing blanks count as characters. If the character you want to remove is preceded (for leading) or followed (for trailing) by a blank, the character will not be removed. Alphanumeric fields that are longer than the number of characters stored within them are padded with trailing blanks.

- The function will be optimized when run against a relational DBMS that supports trimming the character and location specified.

Syntax: **How to Remove a Leading Character, Trailing Character, or Both From a String**

```
TRIM_(where, pattern, string)
```

where:

where

Keyword

Defines where to trim the source string. Valid values are:

- **LEADING,** which removes leading occurrences.
- **TRAILING,** which removes trailing occurrences.
- **BOTH,** which removes leading and trailing occurrences.

pattern

Alphanumeric

Is a single character, enclosed in single quotation marks ('), whose occurrences are to be removed from *string*. For example, the character can be a single blank (' ').

string

Alphanumeric

Is the string to be trimmed.

The data type of the returned string is AnV.

Example: **Trimming a Character From a String**

TRIM_ removes leading occurrences of the character 'B' from DIRECTOR.

```
TRIM_(LEADING, 'B', DIRECTOR)
```

For BROOKS R., the result is ROOKS R.

UPPER: Returning a String With All Letters Uppercase

The UPPER function takes a source string and returns a string of the same data type with all letters translated to uppercase.

Syntax: How to Return a String With All Letters Uppercase

```
UPPER(string)
```

where:

`string`

Alphanumeric

Is the string to convert to uppercase.

The returned string is the same data type and length as the source string.

Example: Converting Letters to Uppercase

LAST_NAME_MIXED has the last name in mixed case. UPPER converts LAST_NAME_MIXED to uppercase.

```
UPPER(LAST_NAME_MIXED)
```

For Banning , the result is BANNING.

Chapter 4

Character Functions

Character functions manipulate alphanumeric fields and character strings.

In this chapter:

ARGLEN: Measuring the Length of a String

The ARGLEN function measures the length of a character string within a field, excluding trailing spaces. The field format in a Master File specifies the length of a field, including trailing spaces.

Syntax: How to Measure the Length of a Character String

```
ARGLEN( length, source_string, output)
```

where:

`length`
Integer

Is the length of the field containing the character string, or a field that contains the length.

`source_string`
Alphanumeric

Is the name of the field containing the character string.

`output`
Integer

Example: Measuring the Length of a Character String

ARGLEN determines the length of the character string in LAST_NAME and stores the result in a column with the format I3:

```
ARGLEN(15, LAST_NAME, 'I3')
```

For SMITH, the result is 5.

For BLACKWOOD, the result is 9.

ASIS: Distinguishing Between Space and Zero

The ASIS function distinguishes between a space and a zero in Dialogue Manager. It differentiates between a numeric string, a constant or variable defined as a numeric string (number within single quotation marks), and a field defined simply as numeric. ASIS forces a variable to be evaluated as it is entered rather than be converted to a number. It is used in Dialogue Manager equality expressions only.

Syntax: How to Distinguish Between a Space and a Zero

```
ASIS(argument)
```

where:

argument

Alphanumeric

Is the value to be evaluated.

If you specify an alphanumeric literal, enclose it in single quotation marks. If you specify an expression, use parentheses, as needed, to ensure the correct order of evaluation.

Example: Distinguishing Between a Space and a Zero

The first request does not use ASIS. No difference is detected between variables defined as a space and 0.

```
-SET &VAR1 = ' ';
-SET &VAR2 = 0;
-IF &VAR2 EQ &VAR1 GOTO ONE;
-TYPE VAR1 &VAR1 EQ VAR2 &VAR2 NOT TRUE
-QUIT
-ONE
-TYPE VAR1 &VAR1 EQ VAR2 &VAR2 TRUE
```

The output is:

```
VAR1 EQ VAR2 0 TRUE
```

The next request uses ASIS to distinguish between the two variables.

```
-SET &VAR1 = ' ';
-SET &VAR2 = 0;
-IF &VAR2 EQ ASIS(&VAR1) GOTO ONE;
-TYPE VAR1 &VAR1 EQ VAR2 &VAR2 NOT TRUE
-QUIT
-ONE
-TYPE VAR1 &VAR1 EQ VAR2 &VAR2 TRUE
```

The output is:

```
VAR1 EQ VAR2 0 NOT TRUE
```

Reference: Usage Notes for ASIS

In general, Dialogue Manager variables are treated as alphanumeric values. However, a Dialogue Manager variable with the value of '.' may be treated as an alphanumeric value ('.') or a number (0) depending on the context used.

- If the Dialogue Manager variable '.' is used in a mathematical expression, its value will be treated as a number. For example, in the following request, &DMVAR1 is used in an arithmetic expression and is evaluated as zero (0).

  ```
  -SET &DMVAR1='.';
  -SET &DMVAR2=10 + &DMVAR1;
  -TYPE DMVAR2 = &DMVAR2
  ```

  The output is;

  ```
  DMVAR2 = 10
  ```

- If the Dialogue Manager variable value '.' is used in an IF test and is compared to the values ' ', '0', or '.', the result will be TRUE even if ASIS is used, as shown in the following example. The following IF tests all evaluate to TRUE.

  ```
  -SET &DMVAR1='.';
  -SET &DMVAR2=IF &DMVAR1 EQ ' ' THEN 'TRUE' ELSE 'FALSE';
  -SET &DMVAR3=IF &DMVAR1 EQ '.' THEN 'TRUE' ELSE 'FALSE';
  -SET &DMVAR4=IF &DMVAR1 EQ '0' THEN 'TRUE' ELSE 'FALSE';
  ```

- If the Dialogue Manager variable is used with ASIS, the result of the ASIS function will be always be considered alphanumeric and will distinguish between the space (‘ ‘), zero (‘0’), or period (‘.’), as in the following example. The following IF tests all evaluate to TRUE.

  ```
  -SET &DMVAR2=IF ASIS('.') EQ '.' THEN 'TRUE' ELSE 'FALSE';
  -SET &DMVAR3=IF ASIS(' ') EQ ' ' THEN 'TRUE' ELSE 'FALSE';
  -SET &DMVAR4=IF ASIS('0') EQ '0' THEN 'TRUE' ELSE 'FALSE';
  ```

- Comparing ASIS('0') to ' ' and ASIS(' ') to '0' always evaluates to FALSE.

BITSON: Determining If a Bit Is On or Off

The BITSON function evaluates an individual bit within a character string to determine whether it is on or off. If the bit is on, BITSON returns a value of 1. If the bit is off, it returns a value of 0. This function is useful in interpreting multi-punch data, where each punch conveys an item of information.

Syntax: How to Determine If a Bit Is On or Off

```
BITSON(bitnumber, source_string, output)
```

where:

bitnumber

Integer

Is the number of the bit to be evaluated, counted from the left-most bit in the character string.

source_string

Alphanumeric

Is the character string to be evaluated. The character string is in multiple eight-bit blocks.

output

Integer

Is the name of the field that contains the result, or the format of the output value enclosed in single quotation marks.

Example: Evaluating a Bit in a Field

BITSON evaluates the 24th bit of LAST_NAME:

```
BITSON(24, LAST_NAME, 'I1')
```

For SMITH, the result is 1.

For CROSS, the result is 9.

BITVAL: Evaluating a Bit String as an Integer

The BITVAL function evaluates a string of bits within a character string. The bit string can be any group of bits within the character string and can cross byte and word boundaries. The function evaluates the subset of bits in the string as an integer value.

If the number of bits is:

- ❑ Less than 1, the returned value is 0.
- ❑ Between 1 and 31 (the recommended range), the returned value is a zero or positive number representing the bits specified, extended with high-order zeroes for a total of 32 bits.
- ❑ Exactly 32, the returned value is the positive, zero, or the complement value of negative two, of the specified 32 bits.

- ❏ Greater than 32 (33 or more), the returned value is the positive, zero, or the complement value of negative two, of the rightmost 32 bits specified.

Syntax: **How to Evaluate a Bit String**

```
BITVAL(source_string, startbit, number, output)
```

where:

`source_string`

Alphanumeric

Is the character string to be evaluated.

`startbit`

Integer

Is the number of the first bit in the bit string, counting from the left-most bit in the character string. If this argument is less than or equal to 0, the function returns a value of zero.

`number`

Integer

Is the number of bits in the subset of bits. If this argument is less than or equal to 0, the function returns a value of zero.

`output`

Integer

Example: **Evaluating a Bit String**

BITVAL evaluates the bits 12 through 20 of LAST_NAME and stores the result in a column with the format I5:

```
BITVAL(LAST_NAME, 12, 9, 'I5')
```

For SMITH, the result is 332.

For JONES, the result is 365.

BYTVAL: Translating a Character to Decimal

The BYTVAL function translates a character to the ASCII, EBCDIC, or Unicode decimal value that represents it, depending on the operating system.

Syntax: **How to Translate a Character**

```
BYTVAL(character, output)
```

where:

character

Alphanumeric

Is the character to be translated. If you supply more than one character, the function evaluates the first.

output

Integer

Example: **Translating the First Character of a Field**

BYTVAL translates the first character of LAST_NAME into its ASCII decimal value and stores the result in a column with the format I3.

```
BYTVAL(LAST_NAME,'I3')
```

For SMITH, the result is 83.

For JONES the result is 74.

CHKFMT: Checking the Format of a String

The CHKFMT function checks a character string for incorrect characters or character types. It compares each character string to a second string, called a mask, by comparing each character in the first string to the corresponding character in the mask. If all characters in the character string match the characters or character types in the mask, CHKFMT returns the value 0. Otherwise, CHKFMT returns a value equal to the position of the first character in the character string not matching the mask.

If the mask is shorter than the character string, the function checks only the portion of the character string corresponding to the mask. For example, if you are using a four-character mask to test a nine-character string, only the first four characters in the string are checked; the rest are returned as a no match with CHKFMT giving the first non-matching position as the result.

Syntax: How to Check the Format of a Character String

```
CHKFMT(numchar, source_string, 'mask', output)
```

where:

numchar

Integer

Is the number of characters being compared to the mask.

string

Alphanumeric

Is the character string to be checked.

'mask'

Alphanumeric

Is the mask, which contains the comparison characters enclosed in single quotation marks.

Some characters in the mask are generic and represent character types. If a character in the string is compared to one of these characters and is the same type, it matches. Generic characters are:

`A` is any letter between A and Z (uppercase or lowercase).

`9` is any digit between 0–9.

`X` is any letter between A–Z or any digit between 0-9.

`$` is any character.

Any other character in the mask represents only that character. For example, if the third character in the mask is B, the third character in the string must be B to match.

output

Integer

Example: Checking the Format of a Field

CHKFMT examines EMP_ID for nine numeric characters starting with 11 and stores the result in a column with the format I3.

```
CHKFMT(9, EMP_ID, '119999999', 'I3')
```

For 071382660, the result is 1.

For 119265415, the result is 0.

For 23764317, the result is 2.

CHKNUM: Checking a String for Numeric Format

The CHKNUM function checks a character string for numeric format. If the string contains a valid numeric format, CHKNUM returns the value 1. If the string contains characters that are not valid in a number, CHKNUM returns zero (0).

Syntax: How to Check the Format of a Character String

```
CHKNUM(numchar, source_string, output)
```

where:

numchar

Integer

Is the number of characters in the string.

string

Alphanumeric

Is the character string to be checked.

output

Numeric

Example: Checking a String for Numeric Format

CHKNUM examines STR1 for numeric format.

```
CHKNUM(8, str1, 'I1')
```

For 12345E01, the result is 1.

For ABCDEFG, the result is 0.

CTRAN: Translating One Character to Another

The CTRAN function translates a character within a character string to another character based on its decimal value. This function is especially useful for changing replacement characters to unavailable characters, or to characters that are difficult to input or unavailable on your keyboard. It can also be used for inputting characters that are difficult to enter when responding to a Dialogue Manager -PROMPT command, such as a comma or apostrophe. It eliminates the need to enclose entries in single quotation marks (').

To use CTRAN, you must know the decimal equivalent of the characters in internal machine representation. Note that the coding chart for conversion is platform dependent, hence your platform and configuration option determines whether ASCII, EBCDIC, or Unicode coding is used. Printable EBCDIC or ASCII characters and their decimal equivalents are listed in *Character Chart for ASCII and EBCDIC* on page 18.

In Unicode configurations, this function uses values in the range:

- 0 to 255 for 1-byte characters.
- 256 to 65535 for 2-byte characters.
- 65536 to 16777215 for 3-byte characters.
- 16777216 to 4294967295 for 4-byte characters (primarily for EBCDIC).

Syntax:

How to Translate One Character to Another

```
CTRAN(length, source_string, decimal, decvalue, output)
```

where:

length

Integer

Is the number of characters in the source string,.

source_string

Alphanumeric

Is the character string to be translated.

decimal

Integer

Is the ASCII or EBCDIC decimal value of the character to be translated.

decvalue

Integer

Is the ASCII or EBCDIC decimal value of the character to be used as a substitute for *decimal*.

output

Alphanumeric

Example: Translating Spaces to Underscores on an ASCII Platform

CTRAN translates the spaces in ADDRESS_LN3 (ASCII decimal value of 32) to underscores (ASCII decimal value of 95) and stores the result in a column with the format A20.

```
CTRAN(20, PRODNAME, 32, 95, 'A20')
```

For RUTHERFORD NJ 07073, the result is RUTHERFORD_NJ_07073_.

For NEW YORK NY 10039, the result is NEW_YORK_NY_10039___.

CTRFLD: Centering a Character String

The CTRFLD function centers a character string within a field. The number of leading spaces is equal to or one less than the number of trailing spaces.

CTRFLD is useful for centering the contents of a field and its report column, or a heading that consists only of an embedded field. HEADING CENTER centers each field value including trailing spaces. To center the field value without the trailing spaces, first center the value within the field using CTRFLD.

Syntax: How to Center a Character String

```
CTRFLD(source_string, length, output)
```

where:

source_string
Alphanumeric

Is the character string enclosed in single quotation marks, or a field or variable that contains the character string.

length
Integer

Is the number of characters in *source_string* and *output*. This argument must be greater than 0. A length less than 0 can cause unpredictable results.

output
Alphanumeric

Example: Centering a Field

CTRFLD centers LAST_NAME and stores the result in a column with the format A12:

```
CTRFLD(LAST_NAME, 12, 'A12')
```

EDIT: Extracting or Adding Characters

The EDIT function extracts characters from the source string and adds characters to the output string, according to the mask. It can extract a substring from different parts of the source string. It can also insert characters from the source string into an output string. For example, it can extract the first two characters and the last two characters of a string to form a single output string.

EDIT compares the characters in a mask to the characters in a source string. When it encounters a nine (9) in the mask, EDIT copies the corresponding character from the source field to the output string. When it encounters a dollar sign ($) in the mask, EDIT ignores the corresponding character in the source string. When it encounters any other character in the mask, EDIT copies that character to the corresponding position in the output string. This process ends when the mask is exhausted.

Note:

- ❏ EDIT does not require an output argument because the result is alphanumeric and its size is determined from the mask value.
- ❏ EDIT can also convert the format of a field. For information on converting a field with EDIT, see *EDIT: Converting the Format of a Field* on page 296.

Syntax: How to Extract or Add Characters

```
EDIT(source_string, 'mask');
```

where:

source_string

Alphanumeric

Is a character string from which to pick characters. Each 9 in the mask represents one digit, so the size of *source_string* must be at least as large as the number of 9's in the mask.

mask

Alphanumeric

Is a string of mask characters enclosed in single quotation marks. The length of the mask, excluding characters other than 9 and $, determines the length of the output field.

Example: Extracting Characters

EDIT extracts the first initials from the FNAME column.

```
EDIT(FNAME, '9$$$$$$$$$')
```

For GREGORY, the result is G.

For STEVEN, the result is S.

GETTOK: Extracting a Substring (Token)

The GETTOK function divides a character string into substrings, called tokens. The data must have a specific character, called a delimiter, that occurs in the string and separates the string into tokens. GETTOK returns the token specified by the *token_number* argument. GETTOK ignores leading and trailing blanks in the source character string.

For example, suppose you want to extract the fourth word from a sentence. In this case, use the space character for a delimiter and the number 4 for *token_number*. GETTOK divides the sentence into words using this delimiter, then extracts the fourth word. If the string is not divided by the delimiter, use the PARAG function for this purpose. See *PARAG: Dividing Text Into Smaller Lines* on page 129.

Syntax: How to Extract a Substring (Token)

```
GETTOK(source_string, inlen, token_number, 'delim', outlen, output)
```

where:

source_string

Alphanumeric

Is the source string from which to extract the token.

inlen

Integer

Is the number of characters in *source_string*. If this argument is less than or equal to 0, the function returns spaces.

token_number

Integer

Is the number of the token to extract. If this argument is positive, the tokens are counted from left to right. If this argument is negative, the tokens are counted from right to left. For example, -2 extracts the second token from the right. If this argument is 0, the function returns spaces. Leading and trailing null tokens are ignored.

'delim'

Alphanumeric

Is the delimiter in the source string enclosed in single quotation marks. If you specify more than one character, only the first character is used.

outlen

Integer

Is the size of the token extracted. If this argument is less than or equal to 0, the function returns spaces. If the token is longer than this argument, it is truncated; if it is shorter, it is padded with trailing spaces.

output

Alphanumeric

Note that the delimiter is not included in the extracted token.

Example: **Extracting a Token**

GETTOK extracts the last token from ADDRESS_LN3 and stores the result in a column with the format A10:

```
GETTOK(ADDRESS_LN3, 20, -1, ' ', 10, 'A10')
```

In this case, the last token will be the ZIP code.

For RUTHERFORD NJ 07073, the result is 07073.

For NEW YORK NY 10039, the result is 10039.

LCWORD: Converting a String to Mixed-Case

The LCWORD function converts the letters in a character string to mixed-case. It converts every alphanumeric character to lowercase except the first letter of each new word and the first letter after a single or double quotation mark, which it converts to uppercase. For example, O'CONNOR is converted to O'Connor and JACK'S to Jack'S.

LCWORD skips numeric and special characters in the source string and continues to convert the following alphabetic characters. The result of LCWORD is a string in which the initial uppercase characters of all words are followed by lowercase characters.

Syntax: How to Convert a Character String to Mixed-Case

```
LCWORD(length, source_string, output)
```

where:

`length`

Integer

Is the number of characters in *source_string* and *output*.

`string`

Alphanumeric

Is the character string to be converted.

`output`

Alphanumeric

Example: Converting a Character String to Mixed-Case

LCWORD converts LAST_NAME to mixed-case and stores the result in a column with the format A15:

```
LCWORD(15, LAST_NAME, 'A15')
```

For STEVENS, the result is Stevens.

For SMITH, the result is Smith.

LCWORD2: Converting a String to Mixed-Case

The LCWORD2 function converts the letters in a character string to mixed-case by converting the first letter of each word to uppercase and converting every other letter to lowercase. In addition, a double quotation mark or a space indicates that the next letter should be converted to uppercase.

For example, "SMITH" would be changed to "Smith" and "JACK S" would be changed to "Jack S".

Syntax: **How to Convert a Character String to Mixed-Case**

```
LCWORD2(length, string, output)
```

where:

length

Integer

Is the length, in characters, of the character string or field to be converted, or a field that contains the length.

string

Alphanumeric

Is the character string to be converted, or a temporary field that contains the string.

output

Alphanumeric

The length must be greater than or equal to *length*.

Example: **Converting a Character String to Mixed-Case**

LCWORD2 converts the string O'CONNOR’s to mixed-case:

The value returned is O’Connor’s.

LCWORD3: Converting a String to Mixed-Case

The LCWORD3 function converts the letters in a character string to mixed-case by converting the first letter of each word to uppercase and converting every other letter to lowercase. In addition, a single quotation mark indicates that the next letter should be converted to uppercase, as long as it is neither followed by a blank nor the last character in the input string.

For example, 'SMITH' would be changed to 'Smith' and JACK'S would be changed to Jack's.

Syntax: **How to Convert a Character String to Mixed-Case Using LCWORD3**

```
LCWORD3(length, string, output)
```

where:

length

Integer

Is the length, in characters, of the character string or field to be converted, or a field that contains the length.

string

Alphanumeric

Is the character string to be converted, or a field that contains the string.

output

Alphanumeric

The length must be greater than or equal to *length*.

Example: **Converting a Character String to Mixed-Case Using LCWORD3**

For the string O'CONNOR's, LCWORD3 returns O'Connor's.

For the string o'connor's, LCWORD3 also returns O'Connor's.

LJUST: Left-Justifying a String

LJUST left-justifies a character string.

Syntax: **How to Left-Justify a Character String**

```
LJUST(length, source_string, output)
```

where:

length

Integer

Is the number of characters in *source_string* and *output*.

source_string

Alphanumeric

Is the character string to be justified.

output

Alphanumeric

Example: **Left-Justifying a String**

LJUST left-justifies FNAME and stores the result in a column with the format A25:

```
LJUST(15, FNAME, 'A25')
```

LOCASE: Converting Text to Lowercase

The LOCASE function converts alphanumeric text to lowercase.

Syntax: How to Convert Text to Lowercase

```
LOCASE( length, source_string, output)
```

where:

length

Integer

Is the number of characters in *source_string* and *output*. The length must be greater than 0 .

source_string

Alphanumeric

Is the character string to convert.

output

Alphanumeric

Example: Converting a String to Lowercase

LOCASE converts LAST_NAME to lowercase and stores the result in a column with the format A15:

```
LOCASE(15, LAST_NAME, 'A15')
```

For SMITH, the result is smith.

For JONES, the result is jones.

OVRLAY: Overlaying a Character String

The OVRLAY function overlays a base character string with a substring. The function enables you to edit part of an alphanumeric field without replacing the entire field.

Syntax: How to Overlay a Character String

```
OVRLAY(source_string, length, substring, sublen, position, output)
```

where:

source_string

Alphanumeric

Is the base character string.

`stringlen`

Integer

Is the number of characters in *source_string* and *output*. If this argument is less than or equal to 0, unpredictable results occur.

`substring`

Alphanumeric

Is the substring that will overlay *source_string*.

`sublen`

Integer

Is the number of characters in *substring*. If this argument is less than or equal to 0, the function returns spaces.

`position`

Integer

Is the position in *source_string* at which the overlay begins. If this argument is less than or equal to 0, the function returns spaces. If this argument is larger than *stringlen*, the function returns the source string.

`output`

Alphanumeric

Note that if the overlaid string is longer than the output field, the string is truncated to fit the field.

Example: Replacing Characters in a Character String

OVRLAY replaces the last three characters of EMP_ID with CURR_JOBCODE to create a new identification code and stores the result in a column with the format A9:

```
OVRLAY(EMP_ID, 9, CURR_JOBCODE, 3, 7, 'A9')
```

For EMP_ID of 326179357 with CURR_JOBCODE of B04, the result is 26179B04.

For EMP_ID of 818692173 with CURR_JOBCODE of A17, the result is 818692A17.

PARAG: Dividing Text Into Smaller Lines

The PARAG function divides a character string into substrings by marking them with a delimiter. It scans a specific number of characters from the beginning of the string and replaces the last space in the group scanned with the delimiter, thus creating a first substring, also known as a token. It then scans the next group of characters in the line, starting from the delimiter, and replaces its last space with a second delimiter, creating a second token. It repeats this process until it reaches the end of the line.

Once each token is marked off by the delimiter, you can use the function GETTOK to place the tokens into different fields (see *GETTOK: Extracting a Substring (Token)* on page 123). If PARAG does not find any spaces in the group it scans, it replaces the first character after the group with the delimiter. Therefore, make sure that any group of characters has at least one space. The number of characters scanned is provided as the maximum token size.

For example, if you have a field called 'subtitle' which contains a large amount of text consisting of words separated by spaces, you can cut the field into roughly equal substrings by specifying a maximum token size to divide the field. If the field is 350 characters long, divide it into three substrings by specifying a maximum token size of 120 characters. This technique enables you to print lines of text in paragraph form.

Tip: If you divide the lines evenly, you may create more sub-lines than you intend. For example, suppose you divide 120-character text lines into two lines of 60 characters maximum, but one line is divided so that the first sub-line is 50 characters and the second is 55. This leaves room for a third sub-line of 15 characters. To correct this, insert a space (using weak concatenation) at the beginning of the extra sub-line, then append this sub-line (using strong concatenation) to the end of the one before it. Note that the sub-line will be longer than 60 characters.

Syntax: How to Divide Text Into Smaller Lines

```
PARAG(length, source_string, 'delimiter', max_token_size, output)
```

where:

length

Integer

Is the number of characters in *source_string* and *output*.

source_string

Alphanumeric

Is a string to divide into tokens.

delimiter

Alphanumeric

Is the delimiter enclosed in single quotation marks. Choose a character that does not appear in the text.

max_token_size

Integer

Is the upper limit for the size of each token.

output

Alphanumeric

Example: **Dividing Text Into Smaller Lines**

PARAG divides ADDRESS_LN2 into smaller lines of not more than ten characters, using a comma as the delimiter. The result is stored in a column with the format A20:

```
PARAG(20, ADDRESS_LN2, ',', 10, 'A20')
```

For 147-15 NORTHERN BLD, the result is 147-15,NORTHERN,BLD.

For 13 LINDEN AVE., the result is 13 LINDEN,AVE.

PATTERN: Generating a Pattern From a String

The PATTERN function examines a source string and produces a pattern that indicates the sequence of numbers, uppercase letters, and lowercase letters in the source string. This function is useful for examining data to make sure that it follows a standard pattern.

In the output pattern:

- ❑ Any character from the input that represents a single-byte digit becomes the character *9*.
- ❑ Any character that represents an uppercase letter becomes *A*, and any character that represents a lowercase letter becomes *a*. For European NLS mode (Western Europe, Central Europe), *A* and *a* are extended to apply to accented alphabets.
- ❑ For Japanese, double-byte characters and Hankaku-katakana become *C* (uppercase). Note that double-byte includes Hiragana, Katakana, Kanji, full-width alphabets, full-width numbers, and full-width symbols. This means that all double-byte letters such as Chinese and Korean are also represented as *C*.
- ❑ Special characters remain unchanged.
- ❑ An unprintable character becomes the character *X*.

Syntax: **How to Generate a Pattern From an Input String**

```
PATTERN (length, source_string,  output)
```

where:

length

Numeric

Is the length of *source_string*.

source_string
Alphanumeric

Is the source string.

output
Alphanumeric

Example: **Producing a Pattern From Alphanumeric Data**

PATTERN generates a pattern for each instance of TESTFLD. The result is stored in a column with the format A14:

```
PATTERN (14, TESTFLD, 'A14' )
```

For 212-736-6250, the result is 999-999-9999.

For 800-969-INFO, the result is 1999-999-AAAA.

POSIT: Finding the Beginning of a Substring

The POSIT function finds the starting position of a substring within a source string. For example, the starting position of the substring DUCT in the string PRODUCTION is 4. If the substring is not in the parent string, the function returns the value 0.

Syntax: **How to Find the Beginning of a Substring**

```
POSIT(source_string, length, substring, sublength, output)
```

where:

source_string
Alphanumeric

Is the string to parse.

length
Integer

Is the number of characters in the source string. If this argument is less than or equal to 0, the function returns a 0.

substring
Alphanumeric

Is the substring whose position you want to find.

`sublength`

Integer

Is the number of characters in *substring*. If this argument is less than or equal to 0, or if it is greater than *length*, the function returns a 0.

`output`

Integer

Example: **Finding the Position of a Letter**

POSIT determines the position of the first capital letter I in LAST_NAME and stores the result in a column with the format I2:

```
POSIT(LAST_NAME, 15, 'I', 1, 'I2')
```

For STEVENS, the result is 0.

For SMITH, the result is 3.

For IRVING, the result is 1.

REVERSE: Reversing the Characters in a String

The REVERSE function reverses the characters in a string.

Syntax: **How to Reverse the Characters in a String**

```
REVERSE(length, source_string, output)
```

where:

`length`

Integer

Is the number of characters in *source_string* and *output*.

`source_string`

Alphanumeric

Is the character string to reverse.

`output`

Alphanumeric

Example: Reversing the Characters in a String

REVERSE reverses the characters in PRODCAT and stores the result in a column with the format A15:

```
REVERSE(15, PRODCAT, 'A15')
```

For VCRs, the result is sRCV.

For DVD, the result is DVD.

RJUST: Right-Justifying a Character String

The RJUST function right-justifies a character string. All trailing blacks become leading blanks. This is useful when you display alphanumeric fields containing numbers.

Syntax: How to Right-Justify a Character String

```
RJUST(length, source_string, output)
```

where:

length

Integer

Is the number of characters in *source_string* and *output* Their lengths must be the same to avoid justification problems.

source_string

Alphanumeric

Is the character string to right justify.

output

Alphanumeric

Example: Right-Justifying a String

RJUST right-justifies LAST_NAME and stores the result in a column with the format A15:

```
RJUST(15, LAST_NAME, 'A15')
```

SOUNDEX: Comparing Character Strings Phonetically

The SOUNDEX function analyzes a character string phonetically, without regard to spelling. It converts character strings to four character codes. The first character must be the first character in the string. The last three characters represent the next three significant sounds in the source string.

Syntax: How to Compare Character Strings Phonetically

```
SOUNDEX( length, source_string, output)
```

where:

`length`

Alphanumeric

Is the number of characters in *source_string*. The number must be from 01 to 99, expressed with two digits (for example '01'); a number larger than 99 causes the function to return asterisks (*) as output.

`source_string`

Alphanumeric

Is the string to analyze.

`output`

Alphanumeric

Example: Comparing Character Strings Phonetically

SOUNDEX analyzes LAST_NAME phonetically and stores the result in a column with the format A4.

```
SOUNDEX('15', LAST_NAME, 'A4')
```

SPELLNM: Spelling Out a Dollar Amount

The SPELLNM function spells out an alphanumeric string or numeric value containing two decimal places as dollars and cents. For example, the value 32.50 is THIRTY TWO DOLLARS AND FIFTY CENTS.

Syntax: How to Spell Out a Dollar Amount

```
SPELLNM(outlength, number, output)
```

where:

`outlength`

Integer

Is the number of characters in *output*.

If you know the maximum value of *number*, use the following table to determine the value of *outlength*:

If number is less than...	...outlength should be
$10	37
$100	45
$1,000	59
$10,000	74
$100,000	82
$1,000,000	96

`number`

Alphanumeric or Numeric (9.2)

Is the number to be spelled out. This value must contain two decimal places.

`output`

Alphanumeric

Example: **Spelling Out a Dollar Amount**

SPELLNM spells out the values in CURR_SAL and stores the result in a column with the format A82:

```
SPELLNM(82, CURR_SAL, 'A82')
```

For $13,200.00, the result is THIRTEEN THOUSAND TWO HUNDRED DOLLARS AND NO CENTS.

For $18,480.00, the result is EIGHTEEN THOUSAND FOUR HUNDRED EIGHTY DOLLARS AND NO CENTS.

SQUEEZ: Reducing Multiple Spaces to a Single Space

The SQUEEZ function reduces multiple contiguous spaces within a character string to a single space. The resulting character string has the same length as the original string but is padded on the right with spaces.

Syntax: How to Reduce Multiple Spaces to a Single Space

```
SQUEEZ( length, source_string, output)
```

where:

length

Integer

Is the number of characters in *source_string* and *output*.

source_string

Alphanumeric

Is the character string to squeeze.

output

Alphanumeric

Example: Reducing Multiple Spaces to a Single Space

SQUEEZ reduces multiple spaces in NAME to a single blank and stores the result in a column with the format A30:

```
SQUEEZ(30, NAME, 'A30')
```

For MARY   SMITH, the result is MARY SMITH.

For DIANE   JONES, the result is DIANE JONES.

For JOHN   MCCOY, the result is JOHN MCCOY.

STRIP: Removing a Character From a String

The STRIP function removes all occurrences of a specific character from a string. The resulting character string has the same length as the original string but is padded on the right with spaces.

Syntax: How to Remove a Character From a String

```
STRIP( length, source_string, char, output)
```

where:

length

Integer

Is the number of characters in *source_string* and *output*.

source_string

Alphanumeric

Is the string from which the character will be removed.

char

Alphanumeric

Is the character to be removed from the string. If more than one character is provided, the left-most character will be used as the strip character.

Note: To remove single quotation marks, use two consecutive quotation marks. You must then enclose this character combination in single quotation marks.

output

Alphanumeric

Example: Removing Occurrences of a Character From a String

STRIP removes all occurrences of a period (.) from DIRECTOR and stores the result in a field with the format A17:

```
STRIP(17, DIRECTOR, '.', 'A17')
```

For ZEMECKIS R., the result is ZEMECKIS R.

For BROOKS J.L., the result is BROOKS JL.

STRREP: Replacing Character Strings

The STRREP replaces all instances of a specified string within a source string. It also supports replacement by null strings.

Syntax: How to Replace Character Strings

```
STRREP (inlength, instring, searchlength, searchstring, replength,
repstring, outlength, output)
```

where:

inlength

Numeric

Is the number of characters in the source string.

instring

Alphanumeric

Is the source string.

searchlength

Numeric

Is the number of characters in the (shorter length) string to be replaced.

searchstring

Alphanumeric

Is the character string to be replaced.

replength

Numeric

Is the number of characters in the replacement string. Must be zero (0) or greater.

repstring

Alphanumeric

Is the replacement string (alphanumeric). Ignored if replength is zero (0).

outlength

Numeric

Is the number of characters in the resulting output string. Must be 1 or greater.

output

Alphanumeric

Reference: Usage Note for STRREP Function

The maximum string length is 4095.

Example: Replacing Commas and Dollar Signs

STRREP finds and replaces commas and then dollar signs and stores the result in field with the format A17:

```
STRREP(15,CS_ALPHA,1,',',0,'X',14,'A14')
STRREP(14,CS_NOCOMMAS,1,'$',4,'USD ',17,'A17')
```

For $29,700.00, the result is USD 29700.00.

For $9,000.00, the result is USD 9000.00.

SUBSTR: Extracting a Substring

The SUBSTR function extracts a substring based on where it begins and its length in the source string.

Syntax: How to Extract a Substring

`SUBSTR(` *`length`*`,` *`source_string`*`,` *`start`*`,` *`end`*`,` *`sublength`*`,` *`output`*`)`

where:

`length`

Integer

Is the number of characters in *source_string*.

`source_string`

Alphanumeric

Is the string from which to extract a substring .

`start`

Integer

Is the starting position of the substring in the source string. If *start* is less than one or greater than *length*, the function returns spaces.

`end`

Integer

Is the ending position of the substring. If this argument is less than *start* or greater than *length*, the function returns spaces.

`sublength`

Integer

Is the number of characters in the substring (normally end - start + 1). If *sublength* is longer than *end* - *start* +1, the substring is padded with trailing spaces. If it is shorter, the substring is truncated. This value should be the declared length of *output*. Only *sublength* characters will be processed.

`output`

Alphanumeric

Example: **Extracting a String**

SUBSTR extracts the first three characters from LAST_NAME, and stores the results in a column with the format A3:

```
SUBSTR(15, LAST_NAME, 1, 3, 3, 'A3')
```

For BANNING, the result is BAN.

For MCKNIGHT, the result is MCK.

TRIM: Removing Leading and Trailing Occurrences

The TRIM function removes leading and/or trailing occurrences of a pattern within a character string.

Syntax: **How to Remove Leading and Trailing Occurrences**

```
TRIM(trim_where, source_string, length, pattern, sublength, output)
```

where:

trim_where

Alphanumeric

Is one of the following, which indicates where to remove the pattern:

'L' removes leading occurrences.

'T' removes trailing occurrences.

'B' removes both leading and trailing occurrences.

source_string

Alphanumeric

Is the string to trim .

string_length

Integer

Is the number of characters in the source string.

pattern

Alphanumeric

Is the character string pattern to remove.

sublength

Integer

Is the number of characters in the pattern.

output

Alphanumeric

Example: **Removing Leading Occurrences**

TRIM removes leading occurrences of the characters BR from DIRECTOR and stores the result in a column with the format A17:

```
TRIM('L', DIRECTOR, 17, 'BR', 2, 'A17')
```

For BROOKS R., the result is OOKS R.

For ABRAHAMS J., the result is ABRAHAMS J.

UPCASE: Converting Text to Uppercase

The UPCASE function converts a character string to uppercase. It is useful for sorting on a field that contains both mixed-case and uppercase values. Sorting on a mixed-case field produces incorrect results because the sorting sequence in EBCDIC always places lowercase letters before uppercase letters, while the ASCII sorting sequence always places uppercase letters before lowercase. To obtain correct results, define a new field with all of the values in uppercase, and sort on that field.

Syntax: **How to Convert Text to Uppercase**

```
UPCASE(length, source_string, output)
```

where:

length

Integer

Is the number of characters in *source_string* and *output*.

input

Alphanumeric

Is the string to convert.

`output`

Alphanumeric of type A*n*V or A*n*

If the format of the output_format is A*n*V, then the length returned is equal to the smaller of the source_string length and the upper_limit length.

Example: **Converting a Mixed-Case String to Uppercase**

UPCASE converts LAST_NAME_MIXED to uppercase and stores the result in a column with the format A15:

```
UPCASE(15, LAST_NAME_MIXED, 'A15')
```

For Banning, the result is BANNING.

For McKnight, the result is MCKNIGHT.

XMLDECOD: Decoding XML-Encoded Characters

The XMLDECOD function decodes the following five standard XML-encoded characters when they are encountered in a string:

Character Name	Character	XML-Encoded Representation
ampersand	&	&amp;
greater than symbol	>	&gt;
less than symbol	<	&lt;
double quotation mark	"	&quot;
single quotation mark (apostrophe)	'	&apos;

Syntax: **How to Decode XML-Encoded Characters**

```
XMLDECOD(inlength, source_string, outlength,  output)
```

where:

`inlength`

Integer

Is the length of the field containing the source character string, or a field that contains the length.

source_string

Alphanumeric

Is the name of the field containing the source character string or the string enclosed in single quotation marks (').

outlength

Integer

Is the length of the output character string, or a field that contains the length.

output

Integer

Example: **Decoding XML-Encoded Characters**

XMLDECOD decodes XML-encoded characters and stores the output in a string with format A30:

```
XMLDECOD(30, INSTRING, 30, 'A30')
```

For &amp;, the result is &.

For &gt;, the result is >.

XMLENCOD: XML-Encoding Characters

The XMLENCOD function encodes the following five standard characters when they are encountered in a string:

Character Name	Character	Encoded Representation
ampersand	&	&amp;
greater than symbol	>	&gt;
less than symbol	<	&lt;
double quotation mark	"	&quot;
single quotation mark (apostrophe)	'	&apos;

Syntax: How to XML-Encode Characters

```
XMLENCOD(inlength, source_string, option, outlength,  output)
```

where:

inlength

Integer

Is the length of the field containing the source character string, or a field that contains the length.

source_string

Alphanumeric

Is the name of the field containing the source character string or a string enclosed in single quotation marks (').

option

Integer

Is a code that specifies whether to process a string that already contains XML-encoded characters. Valid values are:

- ❑ 0, the default, which cancels processing of a string that already contains at least one XML-encoded character.
- ❑ 1, which processes a string that contains XML-encoded characters.

outlength

Integer

Is the length of the output character string, or a field that contains the length.

Note: The output length, in the worst case, could be six times the length of the input.

output

Integer

Example: XML-Encoding Characters

XMLENCOD XML-encodes characters and stores the output in a string with format A30:

```
XMLENCOD(30, INSTRING, 30, 1, 'A30')
```

For &, the result is &amp;.

For >, the result is &gt;.

Chapter 5

Variable Length Character Functions

The character format A*n*V is supported in synonyms for FOCUS, XFOCUS, and relational data sources. This format is used to represent the VARCHAR (variable length character) data types supported by relational database management systems.

In this chapter:

Overview

For relational data sources, A*n*V keeps track of the actual length of a VARCHAR column. This information is especially valuable when the value is used to populate a VARCHAR column in a different RDBMS. It affects whether trailing blanks are retained in string concatenation and, for Oracle, string comparisons (the other relational engines ignore trailing blanks in string comparisons).

In a FOCUS or XFOCUS data source, A*n*V does not provide true variable length character support. It is a fixed-length character field with an extra two leading bytes to contain the actual length of the data stored in the field. This length is stored as a short integer value occupying two bytes. Because of the two bytes of overhead and the additional processing required to strip them, A*n*V format is *not* recommended for use with non-relational data sources.

A*n*V fields can be used as arguments to all Information Builders-supplied functions that expect alphanumeric arguments. An A*n*V input parameter is treated as an A*n* parameter and is padded with blanks to its declared size (*n*). If the last parameter specifies an A*n*V format, the function result is converted to type A*n*V with actual length set equal to its size.

The functions described in this topic are designed to work specifically with the A*n*V data type parameters.

LENV: Returning the Length of an Alphanumeric Field

LENV returns the actual length of an A*n*V field or the size of an A*n* field.

Syntax: How to Find the Length of an Alphanumeric Field

```
LENV(source_string, output)
```

where:

source_string

Alphanumeric of type A*n* or A*n*V

Is the source string or field. If it is an A*n* format field, the function returns its size, *n*. For a character string enclosed in quotation marks or a variable, the size of the string or variable is returned. For a field of A*n*V format, its length, taken from the length-in-bytes of the field, is returned.

output

Integer

Example: Finding the Length of an A*n*V Field

LENV returns the length of TITLEV and stores the result in a column with the format I2:

```
LENV(TITLEV, 'I2')
```

For ALICE IN WONDERLAND, the result is 19.

For SLEEPING BEAUTY, the result is 15.

LOCASV: Creating a Variable Length Lowercase String

The LOCASV function converts alphabetic characters in the source string to lowercase and is similar to LOCASE. LOCASV returns A*n*V output whose actual length is the lesser of the actual length of the A*n*V source string and the value of the input parameter upper_limit.

Syntax: How to Create a Variable Length Lowercase String

```
LOCASV(upper_limit, source_string, output)
```

where:

`upper_limit`

Integer

Is the limit for the length of the source string.

`source_string`

Alphanumeric of type A*n* or A*n*V

Is the string to be converted to lowercase. If it is a field, it can have A*n* or A*n*V format. If it is a field of type A*n*V, its length is taken from the length in bytes stored in the field. If *upper_limit* is smaller than the actual length, the source string is truncated to this upper limit.

`output`

Alphanumeric of type A*n* or A*n*V

If the output format is A*n*V, the actual length returned is equal to the smaller of the source string length and the upper limit.

Example: Creating a Variable Length Lowercase String

LOCASV converts LAST_NAME to lowercase and specifies a length limit of five characters. The results are stored in a column with the format A15V:

```
LOCASV(5, LAST_NAME, 'A15V')
```

For SMITH, the result is smith.

For JONES, the result is jones.

POSITV: Finding the Beginning of a Variable Length Substring

The POSITV function finds the starting position of a substring within a larger string. For example, the starting position of the substring DUCT in the string PRODUCTION is 4. If the substring is not in the parent string, the function returns the value 0. This is similar to POSIT; however, the lengths of its A*n*V parameters are based on the actual lengths of those parameters in comparison with two other parameters that specify their sizes.

Syntax: How to Find the Beginning of a Variable Length Substring

```
POSITV(source_string, upper_limit, substring, sub_limit, output)
```

where:

source_string

Alphanumeric of type A*n* or A*n*V

Is the source string that contains the substring whose position you want to find. If it is a field of A*n*V format, its length is taken from the length bytes stored in the field. If *upper_limit* is smaller than the actual length, the source string is truncated to this upper limit.

upper_limit

Integer

Is a limit for the length of the source string.

substring

Alphanumeric of type A*n* or A*n*V

Is the substring whose position you want to find. If it is a field of type A*n*V, its length is taken from the length bytes stored in the field. If *sub_limit* is smaller than the actual length, the source string is truncated to this limit.

sub_limit

Integer

Is the limit for the length of the substring.

output

Integer

Example: Finding the Starting Position of a Variable Length Pattern

POSITV finds the starting position of a comma in TITLEV, which would indicate a trailing definite or indefinite article in a movie title (such as ", THE" in SMURFS, THE). LENV is used to determine the length of title. The result is stored in a column with the format I4:

```
POSITV(TITLEV,LENV(TITLEV,'I4'), ',', 1,'I4')
```

For “SMURFS, THE”, the result is 7.

For “SHAGGY DOG, THE”, the result is 11.

SUBSTV: Extracting a Variable Length Substring

The SUBSTV function extracts a substring from a string and is similar to SUBSTR. However, the end position for the string is calculated from the starting position and the substring length. Therefore, it has fewer parameters than SUBSTR. Also, the actual length of the output field, if it is an A*n*V field, is determined based on the substring length.

Syntax: How to Extract a Variable Length Substring

```
SUBSTV(upper_limit, source_string, start, sub_limit, output)
```

where:

`upper_limit`

Integer

Is the limit for the length of the source string.

`source_string`

Alphanumeric of type A*n* or A*n*V

Is the character string that contains the substring you want to extract. If it is a field of type A*n*V, its length is taken from the length bytes stored in the field. If *upper_limit* is smaller than the actual length, the source string is truncated to the upper limit. The final length value determined by this comparison is referred to as *p_length* (see the description of the *output* parameter for related information).

`start`

Integer

Is the starting position of the substring in the source string. The starting position can exceed the source string length, which results in spaces being returned.

`sub_limit`

Integer

Is the length, in characters, of the substring. Note that the ending position can exceed the input string length depending on the provided values for *start* and *sub_limit*.

`output`

Alphanumeric of type A*n* or A*n*V

If the format of *output* is A*n*V, and assuming *end* is the ending position of the substring, the actual length, *outlen*, is computed as follows from the values for *end*, *start*, and *p_length* (see the *source_string* parameter for related information):

If *end* > *p_length* or *end* < *start*, then outlen = 0. Otherwise, outlen = *end* - *start* + 1.

Example: Extracting a Variable Length Substring

SUBSTV extracts the first three characters from the TITLEV and stores the result in a column with the format A20V:

```
SUBSTV(39, TITLEV, 1, 3, 'A20V')
```

For SMURFS, the result is SMU.

For SHAGGY DOG, the result is SHA.

TRIMV: Removing Characters From a String

The TRIMV function removes leading and/or trailing occurrences of a pattern within a character string. TRIMV is similar to TRIM. However, TRIMV allows the source string and the pattern to be removed to have A*n*V format.

TRIMV is useful for converting an A*n* field to an A*n*V field (with the length in bytes containing the actual length of the data up to the last non-blank character).

Syntax: How to Remove Characters From a String

```
TRIMV(trim_where, source_string, upper_limit, pattern, pattern_limit,
output)
```

where:

trim_where

Alphanumeric

Is one of the following, which indicates where to remove the pattern:

'L' removes leading occurrences.

'T' removes trailing occurrences.

'B' removes both leading and trailing occurrences.

source_string

Alphanumeric of type A*n* or A*n*V

Is the source string to be trimmed. If it is a field of type A*n*V, its length is taken from the length in bytes stored in the field. If *upper_limit* is smaller than the actual length, the source string is truncated to this upper limit.

upper_limit

Integer

Is the upper limit for the length of the source string.

`pattern`

Alphanumeric of type A*n* or A*n*V

Is the pattern to remove. If it is a field of type A*n*V, its length is taken from the length in bytes stored in the field. If *pattern_limit* is smaller than the actual length, the pattern is truncated to this limit.

`plength_limit`

Integer

Is the limit for the length of the pattern.

`output`

Alphanumeric of type A*n* or A*n*V

If the output format is A*n*V, the length is set to the number of characters left after trimming.

Example: **Creating an A*n*V Field by Removing Trailing Blanks**

TRIMV removes trailing blanks from TITLE and stores the result in a column with the format A39V:

```
TRIMV('T', TITLE, 39, ' ', 1, 'A39V')
```

UPCASV: Creating a Variable Length Uppercase String

UPCASV converts alphabetic characters to uppercase, and is similar to UPCASE. However, UPCASV can return A*n*V output whose actual length is the lesser of the actual length of the A*n*V source string and an input parameter that specifies the upper limit.

Syntax: **How to Create a Variable Length Uppercase String**

```
UPCASV(upper_limit, source_string, output)
```

where:

`upper_limit`

Integer

Is the limit for the length of the source string.

`source_string`

Alphanumeric of type A*n* or A*n*V

is the string to convert to uppercase. If it is a field of type A*n*V, its length is taken from the length in bytes stored in the field. If *upper_limit* is smaller than the actual length, the source string is truncated to the upper limit.

output

Alphanumeric of type A*n* or A*n*V

If the output format is A*n*V, the length returned is equal to the smaller of the source string length and *upper_limit*.

Example: Creating a Variable Length Uppercase String

UPCASEV converts LAST_NAME_MIXED to uppercase and stores the result in a column with the format A15V:

```
UPCASEV(15, LAST_NAME_MIXED, 'A15V5')
```

For Banning, the result is BANNING.

For McKnight, the result is MCKNIGHT.

Chapter 6

Character Functions for DBCS Code Pages

The functions in this topic manipulate strings of DBCS and SBCS characters when your configuration uses a DBCS code page.

In this chapter:

DCTRAN: Translating A Single-Byte or Double-Byte Character to Another

The DCTRAN function translates a single-byte or double-byte character within a character string to another character based on its decimal value. To use DCTRAN, you need to know the decimal equivalent of the characters in internal machine representation.

The DCTRAN function can translate single-byte to double-byte characters and double-byte to single-byte characters, as well as single-byte to single-byte characters and double-byte to double-byte characters.

Syntax: **How to Translate a Single-Byte or Double-Byte Character to Another**

```
DCTRAN(length, source_string, indecimal, outdecimal, output)
```

where:

length

Double

Is the number of characters in *source_string*.

`source_string`

Alphanumeric

Is the character string to be translated.

`indecimal`

Double

Is the ASCII or EBCDIC decimal value of the character to be translated.

`outdecimal`

Double

Is the ASCII or EBCDIC decimal value of the character to be used as a substitute for *indecimal*.

`output`

Alphanumeric

Example: **Using DCTRAN to Translate Double-Byte Characters**

In the following:

DCTRAN(8, 'AｱA本B語', 177, 70, A8)

For AｱA本B語, the result is AFA本B語.

DEDIT: Extracting or Adding Characters

If your configuration uses a DBCS code page, you can use the DEDIT function to extract characters from or add characters to a string.

DEDIT works by comparing the characters in a mask to the characters in a source field. When it encounters a nine (9) in the mask, DEDIT copies the corresponding character from the source field to the new field. When it encounters a dollar sign ($) in the mask, DEDIT ignores the corresponding character in the source field. When it encounters any other character in the mask, DEDIT copies that character to the corresponding position in the new field.

Syntax: **How to Extract or Add DBCS or SBCS Characters**

```
DEDIT(inlength, source_string, mask_length, mask, output)
```

where:

inlength

Integer

Is the number of *bytes* in *source_string*. The string can have a mixture of DBCS and SBCS characters. Therefore, the number of bytes represents the maximum number of characters possible in the source string.

source_string

Alphanumeric

Is the string to edit.

mask_length

Integer

Is the number of *characters* in mask.

mask

Alphanumeric

Is the string of mask characters.

Each nine (9) in the mask causes the corresponding character from the source field to be copied to the new field.

Each dollar sign ($) in the mask causes the corresponding character in the source field to be ignored.

Any other character in the mask is copied to the new field.

output

Alphanumeric

Example: **Adding and Extracting DBCS Characters**

The following example copies alternate characters from the source string to the new field, starting with the first character in the source string, and then adds several new characters at the end of the extracted string:

```
DEDIT( 15, 'あaいiうuえeおo', 16, '9$9$9$9$9$-かきくけこ', 'A30')
The result is あいうえお-かきくけこ.
```

The following example copies alternate characters from the source string to the new field, starting with the second character in the source string, and then adds several new characters at the end of the extracted string:

```
DEDIT( 15, 'あaいiうuえeおo', 16, '$9$9$9$9$9-ABCDE', 'A20')
The result is aiueo-ABCDE.
```

DSTRIP: Removing a Single-Byte or Double-Byte Character From a String

The DSTRIP function removes all occurrences of a specific single-byte or double-byte character from a string. The resulting character string has the same length as the original string, but is padded on the right with spaces.

Syntax: **How to Remove a Single-Byte or Double-Byte Character From a String**

```
DSTRIP(length, source_string, char, output)
```

where:

length

Double

Is the number of characters in *source_string* and *outfield*.

source_string

Alphanumeric

Is the string from which the character will be removed.

char

Alphanumeric

Is the character to be removed from the string. If more than one character is provided, the left-most character will be used as the strip character.

Note: To remove single quotation marks, use two consecutive quotation marks. You must then enclose this character combination in single quotation marks.

output

Alphanumeric

Example: **Removing a Double-Byte Character From a String**

In the following:

```
DSTRIP(9, 'A日A本B語', '日', A9)
```

For A日A本B語, the result is AA本B語.

DSUBSTR: Extracting a Substring

If your configuration uses a DBCS code page, you can use the DSUBSTR function to extract a substring based on its length and position in the source string.

Syntax: **How to Extract a Substring**

```
DSUBSTR(inlength, source_string, start, end, sublength, output)
```

where:

inlength

Integer

Is the length of the source string in *bytes*. The string can have a mixture of DBCS and SBCS characters. Therefore, the number of bytes represents the maximum number of characters possible in the source string.

source_string

Alphanumeric

Is the string from which the substring will be extracted .

start

Integer

Is the starting position (in number of *characters*) of the substring in the source string. If this argument is less than one or greater than *end*, the function returns spaces.

end

Integer

Is the ending position (in number of *characters*) of the substring. If this argument is less than *start* or greater than *inlength*, the function returns spaces.

sublength

Integer

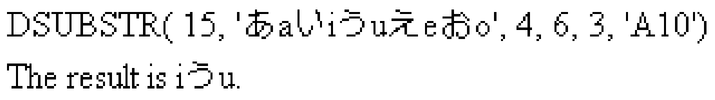
Is the length of the substring, in *characters* (normally *end* - *start* + 1). If *sublength* is longer than *end* - *start* +1, the substring is padded with trailing spaces. If it is shorter, the substring is truncated. This value should be the declared length of *output*. Only *sublength* characters will be processed.

output

Alphanumeric

Example: **Extracting a Substring**

The following example extracts the 3-character substring in positions 4 through 6 from a 15-byte string of characters:



```
DSUBSTR( 15, 'あaいiうuえeおo', 4, 6, 3, 'A10')
The result is iうu.
```

JPTRANS: Converting Japanese Specific Characters

The JPTRANS function converts Japanese specific characters.

Syntax: **How to Convert Japanese Specific Characters**

```
JPTRANS ('type_of_conversion', length, source_string, 'output_format')
```

where:

type_of_conversion

Is one of the following options indicating the type of conversion you want to apply to Japanese specific characters. The following table shows the single component input types:

Conversion Type	Description
`'UPCASE'`	Converts Zenkaku (Fullwidth) alphabets to Zenkaku uppercase.
`'LOCASE'`	Converts Zenkaku alphabets to Zenkaku lowercase.
`'HNZNALPHA'`	Converts alphanumerics from Hankaku (Halfwidth) to Zenkaku.
`'HNZNSIGN'`	Converts ASCII symbols from Hankaku to Zenkaku.

Conversion Type	Description
`'HNZNKANA'`	Converts Katakana from Hankaku to Zenkaku.
`'HNZNSPACE'`	Converts space (blank) from Hankaku to Zenkaku.
`'ZNHNALPHA'`	Converts alphanumerics from Zenkaku to Hankaku.
`'ZNHNSIGN'`	Converts ASCII symbols from Zenkaku to Hankaku.
`'ZNHNKANA'`	Converts Katakana from Zenkaku to Hankaku.
`'ZNHNSPACE'`	Converts space from Zenkaku to Hankaku.
`'HIRAKATA'`	Converts Hiragana to Zenkaku Katakana.
`'KATAHIRA'`	Converts Zenkaku Katakana to Hiragana.
`'930TO939'`	Converts codepage from 930 to 939.
`'939TO930'`	Converts codepage from 939 to 930.

`length`
Integer

Is the number of characters in the source_string.

`source_string`
Alphanumeric

Is the string to convert.

`output_format`
Alphanumeric

Is the name of the field that contains the output, or the format enclosed in single quotation marks (').

Example: **Using the JPTRANS Function**

```
JPTRANS('UPCASE', 20, Alpha_DBCS_Field, 'A20')
```

For ａｂｃ, the result is ＡＢＣ.

```
JPTRANS('LOCASE', 20, Alpha_DBCS_Field, 'A20')
```

For ＡＢＣ, the result is ａｂｃ.

```
JPTRANS('HNZNALPHA', 20, Alpha_SBCS_Field, 'A20')
```

For AaBbCc123, the result is ＡａＢｂＣｃ１２３.

```
JPTRANS('HNZNSIGN', 20, Symbol_SBCS_Field, 'A20')
```

For !@$%,.?, the result is ！＠＄％、。？

```
JPTRANS('HNZNKANA', 20, Hankaku_Katakana_Field, 'A20')
```

For ｢ﾍﾞｰｽﾎﾞｰﾙ｡｣, the result is 「ベースボール。」

```
JPTRANS('HNZNSPACE', 20, Hankaku_Katakana_Field, 'A20')
```

For ｱ ｲ ｳ, the result is ｱ　ｲ　ｳ

```
JPTRANS('ZNHNALPHA', 20, Alpha_DBCS_Field, 'A20')
```

For ＡａＢｂＣｃ１２３, the result is AaBbCc123.

```
JPTRANS('ZNHNSIGN', 20, Symbol_DBCS_Field, 'A20')
```

For ！＠＄％、。？, the result is !@$%,.?

```
JPTRANS('ZNHNKANA', 20, Zenkaku_Katakana_Field, 'A20')
```

For 「ベースボール。」, the result is ｢ﾍﾞｰｽﾎﾞｰﾙ｡｣

```
JPTRANS('ZNHNSPACE', 20, Zenkaku_Katakana_Field, 'A20')
```

For ア　イ　ウ, the result is ア イ ウ

```
JPTRANS('HIRAKATA', 20, Hiragana_Field, 'A20')
```

For あいう, the result is アイウ

```
JPTRANS('KATAHIRA', 20, Zenkaku_Katakana_Field, 'A20')
```

For アイウ, the result is あいう

In the following, codepoints 0x62 0x63 0x64 are converted to 0x81 0x82 0x83, respectively:

```
JPTRANS('930TO939', 20, CP930_Field, 'A20')
```

In the following, codepoints 0x59 0x62 0x63 are converted to 0x81 0x82 0x83, respectively:

```
JPTRANS('939TO930', 20, CP939_Field, 'A20')
```

Reference: Usage Notes for the JPTRANS Function

- HNZNSIGN and ZNHNSIGN focus on the conversion of symbols.

  Many symbols have a one-to-one relation between Japanese Fullwidth characters and ASCII symbols, whereas some characters have one-to-many relations. For example, the Japanese punctuation character (U+3001) and Fullwidth comma , (U+FF0C) will be converted to the same comma , (U+002C). The following EXTRA rule for those special cases is shown below:

  HNZNSIGN:

  - Double Quote " (U+0022) -> Fullwidth Right Double Quote ” (U+201D)
  - Single Quote ' (U+0027) -> Fullwidth Right Single Quote ’ (U+2019)
  - Comma , (U+002C) -> Fullwidth Ideographic Comma (U+3001)
  - Full Stop . (U+002E) -> Fullwidth Ideographic Full Stop ? (U+3002)
  - Backslash \ (U+005C) -> Fullwidth Backslash \ (U+FF3C)
  - Halfwidth Left Corner Bracket (U+FF62) -> Fullwidth Left Corner Bracket (U+300C)
  - Halfwidth Right Corner Bracket (U+FF63) -> Fullwidth Right Corner Bracket (U+300D)
  - Halfwidth Katakana Middle Dot ? (U+FF65) -> Fullwidth Middle Dot · (U+30FB)

  ZNHNSIGN:

  - Fullwidth Right Double Quote ” (U+201D) -> Double Quote " (U+0022)
  - Fullwidth Left Double Quote “ (U+201C) -> Double Quote " (U+0022)
  - Fullwidth Quotation " (U+FF02) -> Double Quote " (U+0022)
  - Fullwidth Right Single Quote ’ (U+2019) -> Single Quote ' (U+0027)

- Fullwidth Left Single Quote ‘ (U+2018) -> Single Quote ' (U+0027)
- Fullwidth Single Quote ' (U+FF07) -> Single Quote ' (U+0027)
- Fullwidth Ideographic Comma (U+3001) -> Comma , (U+002C)
- Fullwidth Comma , (U+FF0C) -> Comma , (U+002C)
- Fullwidth Ideographic Full Stop ? (U+3002) -> Full Stop . (U+002E)
- Fullwidth Full Stop . (U+FF0E) -> Full Stop . (U+002E)
- Fullwidth Yen Sign ¥ (U+FFE5) -> Yen Sign ¥ (U+00A5)
- Fullwidth Backslash \ (U+FF3C) -> Backslash \ (U+005C)
- Fullwidth Left Corner Bracket (U+300C) -> Halfwidth Left Corner Bracket (U+FF62)
- Fullwidth Right Corner Bracket (U+300D) -> Halfwidth Right Corner Bracket (U+FF63)
- Fullwidth Middle Dot · (U+30FB) -> Halfwidth Katakana Middle Dot · (U+FF65)

- HNZNKANA and ZNHNKANA focus on the conversion of Katakana

  They convert not only letters, but also punctuation symbols on the following list:

  - Fullwidth Ideographic Comma (U+3001) <-> Halfwidth Ideographic Comma (U+FF64)
  - Fullwidth Ideographic Full Stop (U+3002) <-> Halfwidth Ideographic Full Stop (U+FF61)
  - Fullwidth Left Corner Bracket (U+300C) <-> Halfwidth Left Corner Bracket (U+FF62)
  - Fullwidth Right Corner Bracket (U+300D) <-> Halfwidth Right Corner Bracket (U+FF63)
  - Fullwidth Middle Dot · (U+30FB) <-> Halfwidth Katakana Middle Dot · (U+FF65)
  - Fullwidth Prolonged Sound (U+30FC) <-> Halfwidth Prolonged Sound (U+FF70)

- JPTRANS can be nested for multiple conversions.

  For example, text data may contain fullwidth numbers and fullwidth symbols. In some situations, they should be cleaned up for ASCII numbers and symbols.

  For バンゴウ＃１２３, the result is バンゴウ#123

```
JPTRANS('ZNHNALPHA', 20, JPTRANS('ZNHNSIGN', 20, Symbol_DBCS_Field,
'A20'), 'A20')
```

- HNZNSPACE and ZNHNSPACE focus on the conversion of a space (blank character).

Currently only conversion between U+0020 and U+3000 is supported.

KKFCUT: Truncating a String

If your configuration uses a DBCS code page, you can use the KKFCUT function to truncate a string.

Syntax:

How to Truncate a String

```
KKFCUT(length, source_string, output)
```

where:

length

Integer

Is the length of the source string in *bytes*. The string can have a mixture of DBCS and SBCS characters. Therefore, the number of bytes represents the maximum number of characters possible in the source string.

source_string

Alphanumeric

Is the string that will be truncated .

output

Alphanumeric

The string will be truncated to the number of bytes in the output field.

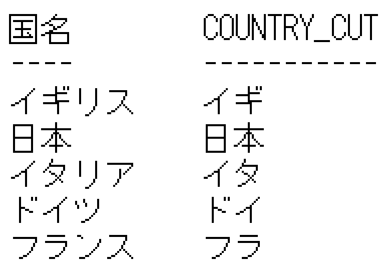
Example:

Truncating a String

In the following, KKFCUT truncates the COUNTRY field (up to 10 bytes long) to A4 format:

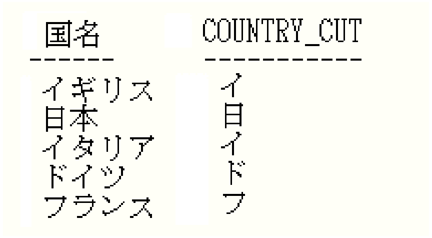
```
COUNTRY_CUT/A4 = KKFCUT(10, COUNTRY, 'A4');
```

The output in ASCII environments is shown in the following image:



```
国名          COUNTRY_CUT
----          -----------
イギリス      イギ
日本          日本
イタリア      イタ
ドイツ        ドイ
フランス      フラ
```

The output in EBCDIC environments is shown in the following image:



```
国名       COUNTRY_CUT
------     -----------
イギリス    イ
日本       日
イタリア    イ
ドイツ      ド
フランス    フ
```

SFTDEL: Deleting the Shift Code From DBCS Data

If your configuration uses a DBCS code page, you can use the SFTDEL function to delete the shift code from DBCS data.

Syntax: How to Delete the Shift Code From DBCS Data

```
SFTDEL(source_string, length, output)
```

where:

source_string
Alphanumeric

Is the string from which the shift code will be deleted .

length
Integer

Is the length of the source string in *bytes*. The string can have a mixture of DBCS and SBCS characters. Therefore, the number of bytes represents the maximum number of characters possible in the source string.

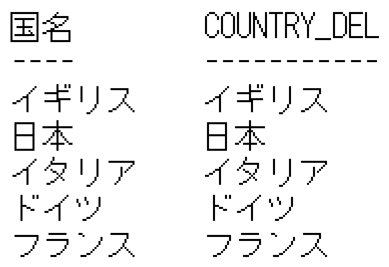
output
Alphanumeric

Example: Deleting the Shift Code From a String

In the following, SFTDEL deleted the shift code from the COUNTRY field (up to 10 bytes long):

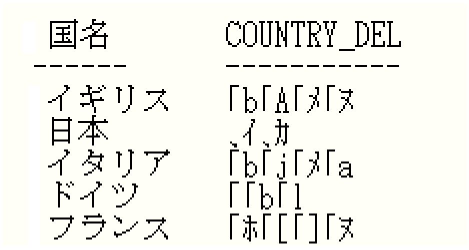
```
COUNTRY_DEL/A10 = SFTDEL(COUNTRY, 10, 'A10');
```

The output in ASCII environments is shown in the following image:



```
国名          COUNTRY_DEL
----          -----------
イギリス      イギリス
日本          日本
イタリア      イタリア
ドイツ        ドイツ
フランス      フランス
```

The output in EBCDIC environments is shown in the following image:



```
 国名          COUNTRY_DEL
-------        -----------
 イギリス      ｢b｢A｢ﾒ｢ｦ
 日本          ､ｲ､ｶ
 イタリア      ｢b｢j｢ﾒ｢a
 ドイツ        ｢｢b｢]
 フランス      ｢ﾎ｢[｢]｢ｦ
```

SFTINS: Inserting the Shift Code Into DBCS Data

If your configuration uses a DBCS code page, you can use the SFTINS function to insert the shift code into DBCS data.

Syntax: **How to Insert the Shift Code Into DBCS Data**

```
SFTINS(source_string, length, output)
```

where:

source_string

Alphanumeric

Is the string into which the shift code will be inserted .

length

Integer

Is the length of the source string in *bytes*. The string can have a mixture of DBCS and SBCS characters. Therefore, the number of bytes represents the maximum number of characters possible in the source string.

output

Alphanumeric

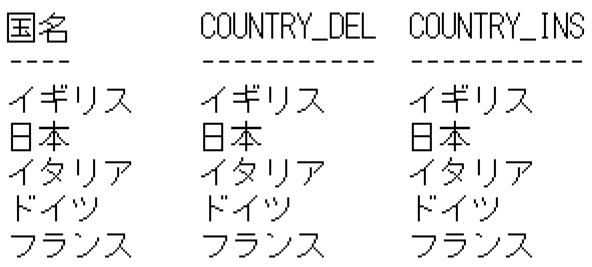
Example: **SFTINS: Inserting the Shift Code Into a String**

In the following example, SFTINS inserts the shift code into the COUNTRY_DEL field (which is the COUNTRY field with the shift code deleted):

```
COUNTRY_INS/A10 = SFTINS(COUNTRY_DEL, 10, 'A10');
```

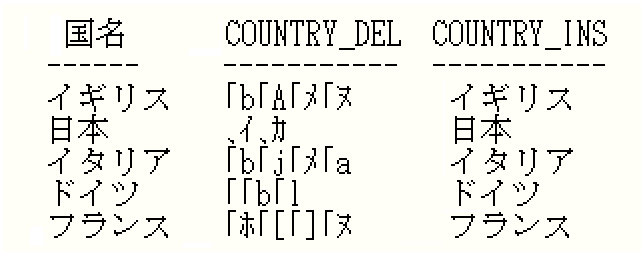
The output displays the original COUNTRY field, the COUNTRY_DEL field with the shift code deleted, and the COUNTRY_INS field with the shift code re-inserted.

The output in ASCII environments, is shown in the following image:



```
国名          COUNTRY_DEL  COUNTRY_INS
----          -----------  -----------
イギリス      イギリス     イギリス
日本          日本         日本
イタリア      イタリア     イタリア
ドイツ        ドイツ       ドイツ
フランス      フランス     フランス
```

The output in EBCDIC environments is shown in the following image:



```
 国名        COUNTRY_DEL  COUNTRY_INS
------       -----------  -----------
イギリス     ｢b｢A｢ﾒ｢ﾇ      イギリス
日本         ､ｲ､ｶ          日本
イタリア     ｢b｢j｢ﾒ｢a      イタリア
ドイツ       ｢｢b｢l         ドイツ
フランス     ｢ﾎ｢[｢]｢ﾇ      フランス
```

Chapter 7

Data Source and Decoding Functions

Data source and decoding functions search for data source records, retrieve data source records or values, and assign values based on the value of an input field.

In this chapter:

CHECKMD5: Computing an MD5 Hash Check Value

CHECKMD5 takes an alphanumeric input value and returns a 128-bit value in a fixed length alphanumeric string, using the MD5 hash function. A hash function is any function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called hash values. They can be used for assuring the integrity of transmitted data.

Syntax: How to Compute an MD5 Hash Check Value

```
CHECKMD5(buffer)
```

where:

buffer

Is a data buffer whose hash value is to be calculated. It can be a set of data of different types presented as a single field, or a group field in one of the following data type formats: An, AnV, or TXn.

Example: Calculating an MD5 Hash Check Value

CHECKMD5 calculates a fixed length MD5 hash check value, and HEXTYPE converts it to a printable hexadecimal string.

```
HEXTYPE(CHECKMD5(PRODUCT_CATEGORY))
```

For Accessories, the result is 98EDB85B00D9527AD5ACEBE451B3FAE6.

CHECKSUM: Computing a Hash Sum

CHECKSUM computes a hash sum, called the checksum, of its input parameter, as a whole number in format I11. This can be used for equality search of the fields. A checksum is a hash sum used to ensure the integrity of a file after it has been transmitted from one storage device to another.

Syntax: How to Compute a CHECKSUM Hash Value

```
CHECKSUM(buffer)
```

where:

buffer

Is a data buffer whose hash index is to be calculated. It can be a set of data of different types presented as a single field, in one of the following data type formats: An, AnV, or TXn.

Example: Calculating a CHECKSUM Hash Value

CHECKSUM calculates a checksum hash value.

```
CHECKSUM(PRODUCT_CATEGORY)
```

For Accessories, the result is -830549649.

COALESCE: Returning the First Non-Missing Value

Given a list of arguments, COALESCE returns the value of the first argument that is not missing. If all argument values are missing, it returns a missing value if MISSING is ON. Otherwise it returns a default value (zero or blank).

Syntax: How to Return the First Non-Missing Value

```
COALESCE(arg1, arg2, ...)
```

where:

`arg1, arg2, ...`

Any field, expression, or constant. The arguments should all be either numeric or alphanumeric.

Are the input parameters that are tested for missing values.

The output data type is the same as the input data types.

Example: Returning the First Non-Missing Value

COALESCE returns the first non-missing value:

```
COALESCE(DAMAGED, RETURNS)
```

The following table shows sample inputs and results.

DAMAGED	RETURNS	RESULT
MISSING	4	4
6	4	6

DB_EXPR: Inserting an SQL Expression Into a Request

The DB_EXPR function inserts a native SQL expression exactly as entered into the native SQL generated for a FOCUS or SQL language request.

The DB_EXPR function can be used in a DEFINE command, a DEFINE in a Master File, a WHERE clause, a FILTER FILE command, a filter in a Master File, or in an SQL statement. It can be used in a COMPUTE command if the request is an aggregate request (uses the SUM, WRITE, or ADD command) and has a single display command. The expression must return a single value.

Syntax: How to Insert an SQL Expression Into a Request With DB_EXPR

```
DB_EXPR(native_SQL_expression)
```

where:

native_SQL_expression

Is a partial native SQL string that is valid to insert into the SQL generated by the request. The SQL string must have double quotation marks (") around each field reference, unless the function is used in a DEFINE with a WITH phrase.

Reference: Usage Notes for the DB_EXPR Function

- ❏ The expression must return a single value.
- ❏ Any request that includes one or more DB_EXPR functions must be for a synonym that has a relational SUFFIX.
- ❏ Field references in the native SQL expression must be within the current synonym context.
- ❏ The native SQL expression must be coded inline. SQL read from a file is not supported.

Example: Inserting the DB2 BIGINT and CHAR Functions Into a TABLE Request

The following TABLE request against the WF_RETAIL data source uses the DB_EXPR function in the COMPUTE command to call two DB2 functions. It calls the BIGINT function to convert the squared revenue to a BIGINT data type, and then uses the CHAR function to convert that value to alphanumeric.

```
TABLE FILE WF
SUM REVENUE NOPRINT
AND COMPUTE BIGREV/A31 = DB_EXPR(CHAR(BIGINT("REVENUE" * "REVENUE") ) ) ;
AS 'Alpha Square Revenue'
BY REGION
ON TABLE SET PAGE NOPAGE
END
```

The trace shows that the expression from the DB_EXPR function was inserted into the DB2 SELECT statement:

```
  SELECT
  T11."REGION",
   SUM(T1."Revenue"),
   ((CHAR(BIGINT( SUM(T1."Revenue") *  SUM(T1."Revenue")) ) ))
   FROM
  wrd_fact_sales T1,
  wrd_dim_customer T5,
  wrd_dim_geography T11
   WHERE
  (T5."ID_CUSTOMER" = T1."ID_CUSTOMER") AND
  (T11."ID_GEOGRAPHY" = T5."ID_GEOGRAPHY")
   GROUP BY
  T11."REGION  "
   ORDER BY
  T11."REGION  "
   FOR FETCH ONLY;
END
```

DB_INFILE: Testing Values Against a File or an SQL Subquery

The DB_INFILE function compares one or more field values in a source file to values in a target file. The comparison can be based on one or more field values. DB_INFILE returns the value 1 (TRUE) if the set of source fields matches a set of values from the target file. Otherwise, the function returns 0 (zero, FALSE). DB_INFILE can be used where a function is valid in a FOCUS request, such as in a DEFINE or a WHERE phrase.

The target file can be any data source that FOCUS can read. Depending on the data sources accessed and the components in the request, either FOCUS or an RDBMS will process the comparison of values.

If FOCUS processes the comparison, it reads the target data source and dynamically creates a sequential file containing the target data values, along with a synonym describing the data file. It then builds IF or WHERE structures in memory with all combinations of source and target values. If the target data contains characters that FOCUS considers wildcard characters, it will treat them as wildcard characters unless the command SET EQTEST = EXACT is in effect.

The following situations exist when a relational data source is the source file:

- **The target values are in a relational data source from the same RDBMS and connection.** In this case, the target file referenced by DB_INFILE can be:
  - An SQL file containing a subquery that retrieves the target values. A synonym must exist that describes the target SQL file. The Access File must specify the CONNECTION and DATASET for the target file.

If the subquery results in a SELECT statement supported by the RDBMS, the relational adapter inserts the subquery into the WHERE predicate of the generated SQL.

If the subquery does not result in a valid SELECT statement for the RDBMS, the relational adapter retrieves the target values. It then generates a WHERE predicate, with a list of all combinations of source and target field values.

You can create an SQL file containing a subquery and a corresponding synonym using the HOLD FORMAT SQL_SCRIPT command.

- A relational data source. A synonym must exist that describes the target data source.

  If the data source contains only those fields referenced by DB_INFILE as target fields, the relational adapter creates a subquery that retrieves the target values. If the subquery results in a SELECT statement supported by the RDBMS, the relational adapter inserts the subquery into the WHERE predicate of the generated SQL.

  If the subquery does not result in a valid SELECT statement for the RDBMS, the relational adapter retrieves a unique list of the target values. It then generates a WHERE predicate with a list of all combinations of source and target field values.

- **The target values are in a non-relational data source or a relational data source from a different RDBMS or connection.** In this case, the target values are retrieved and passed to FOCUS for processing.

Syntax: How to Compare Source and Target Field Values With DB_INFILE

```
DB_INFILE(target_file, s1, t1, ... sn, tn)
```

where:

target_file

Is the synonym for the target file.

s1, ..., sn

Are fields from the source file.

t1, ..., tn

Are fields from the target file.

The function returns the value 1 if a set of target values matches the set of source values. Otherwise, the function returns a zero (0).

Reference: Usage Notes for DB_INFILE

- If both the source and target data sources have MISSING=ON for a comparison field, then a missing value in both files is considered an equality. If MISSING=OFF in one or both files, a missing value in one or both files results in an inequality.
- Values are not padded or truncated when compared, except when comparing date and date-time values.
  - If the source field is a date field and the target field is a date-time field, the time component is removed before comparison.
  - If the source field is a date-time field and the target field is a date field, a zero time component is added to the target value before comparison.
- If an alphanumeric field is compared to a numeric field, an attempt will be made to convert the alphanumeric value to a number before comparison.
- If FOCUS processes the comparison, and the target data contains characters that FOCUS considers wildcard characters, it will treat them as wildcard characters unless the command SET EQTEST = EXACT is in effect.

Example: Comparing Source and Target Values Using an SQL Subquery File

This example uses the WF_RETAIL DB2 data source.

The SQL file named retail_subquery.sql contains the following subquery that retrieves specified state codes in the Central and NorthEast regions:

```
SELECT  MAX(T11.REGION), MAX(T11.STATECODE)  FROM wrd_dim_geography T11
WHERE (T11.STATECODE IN('AR', 'IA', 'KS', 'KY', 'WY', 'CT', 'MA', 'NJ',
'NY', 'RI')) AND (T11.REGION IN('Central', 'NorthEast'))  GROUP BY
T11.REGION, T11.STATECODE
```

The retail_subquery.mas Master File follows:

```
FILENAME=RETAIL_SUBQUERY, SUFFIX=DB2     , $
  SEGMENT=RETAIL_SUBQUERY, SEGTYPE=S0, $
    FIELDNAME=REGION, ALIAS=E01, USAGE=A15V, ACTUAL=A15V,
      MISSING=ON, $
    FIELDNAME=STATECODE, ALIAS=E02, USAGE=A2, ACTUAL=A2,
      MISSING=ON, $
```

The retail_subquery.acx Access File follows:

```
SEGNAME=RETAIL_SUBQUERY, CONNECTION=CON1, DATASET=RETAIL_SUBQUERY.SQL, $
```

The following request uses the DB_INFILE function to compare region names and state codes against the names retrieved by the subquery:

```
TABLE FILE WF
SUM REVENUE
BY REGION
BY STATECODE
WHERE DB_INFILE(RETAIL_SUBQUERY, REGION, REGION, STATECODE, STATECODE)
ON TABLE SET PAGE NOPAGE
END
```

The trace shows that the subquery was inserted into the WHERE predicate in the generated SQL:

```
 SELECT
  T11."REGION",
  T11."STATECODE",
   SUM(T1."Revenue")
   FROM
  wrd_fact_sales T1,
  wrd_dim_customer T5,
  wrd_dim_geography T11
   WHERE
  (T5."ID_CUSTOMER" = T1."ID_CUSTOMER") AND
  (T11."ID_GEOGRAPHY" = T5."ID_GEOGRAPHY") AND
  ((T11."REGION", T11."STATECODE") IN (SELECT  MAX(T11.REGION),
  MAX(T11.STATECODE)  FROM wrd_dim_geography T11 WHERE
  (T11.STATECODE IN('AR', 'IA', 'KS', 'KY', 'WY', 'CT', 'MA',
  'NJ', 'NY', 'RI')) AND (T11.REGION IN('Central', 'NorthEast'))
  GROUP BY T11.REGION, T11.STATECODE))
   GROUP BY
  T11."REGION",
  T11."STATECODE  "
   ORDER BY
  T11."REGION",
  T11."STATECODE  "
   FOR FETCH ONLY;
END
```

Example: **Comparing Source and Target Values Using a Sequential File**

The empvalues.ftm sequential file contains the last and first names of employees in the MIS department:

```
SMITH           MARY        JONES           DIANE       MCCOY
JOHN        BLACKWOOD       ROSEMARIE   GREENSPAN       MARY
CROSS           BARBARA
```

The empvalues.mas Master File describes the data in the empvalues.ftm file

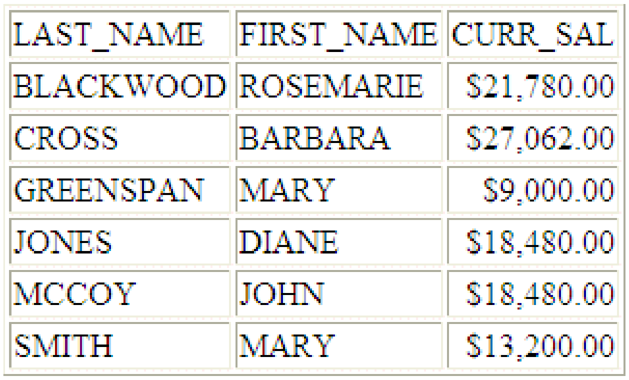
```
FILENAME=EMPVALUES, SUFFIX=FIX     , IOTYPE=BINARY, $
  SEGMENT=EMPVALUE, SEGTYPE=S0, $
    FIELDNAME=LN, ALIAS=E01, USAGE=A15, ACTUAL=A16, $
    FIELDNAME=FN, ALIAS=E02, USAGE=A10, ACTUAL=A12, $
```

Note: You can create a sequential file, along with a corresponding synonym, using the HOLD FORMAT SQL_SCRIPT command.

The following request against the FOCUS EMPLOYEE data source uses the DB_INFILE function to compare employee names against the names stored in the empvalues.ftm file:

```
FILEDEF EMPVALUES DISK baseapp/empvalues.ftm
TABLE FILE EMPLOYEE
SUM CURR_SAL
BY LAST_NAME BY FIRST_NAME
WHERE DB_INFILE(EMPVALUES, LAST_NAME, LN, FIRST_NAME, FN)
ON TABLE SET PAGE NOPAGE
END
```

The output is:



LAST_NAME	FIRST_NAME	CURR_SAL
BLACKWOOD	ROSEMARIE	$21,780.00
CROSS	BARBARA	$27,062.00
GREENSPAN	MARY	$9,000.00
JONES	DIANE	$18,480.00
MCCOY	JOHN	$18,480.00
SMITH	MARY	$13,200.00

Syntax: How to Control DB_INFILE Optimization

To control whether to prevent optimization of the DB_INFILE expression, issue the following command:

```
SET DB_INFILE = {DEFAULT|EXPAND_ALWAYS|EXPAND_NEVER}
```

In a TABLE request, issue the following command:

```
ON TABLE SET DB_INFILE  {DEFAULT|EXPAND_ALWAYS|EXPAND_NEVER}
```

where:

`DEFAULT`

Enables DB_INFILE to create a subquery if its analysis determines that it is possible. This is the default value.

`EXPAND_ALWAYS`

Prevents DB_INFILE from creating a subquery. Instead, it expands the expression into IF and WHERE clauses in memory.

`EXPAND_NEVER`

Prevents DB_INFILE from expanding the expression into IF and WHERE clauses in memory. Instead, it attempts to create a subquery. If this is not possible, a FOC32585 message is generated and processing halts.

DB_LOOKUP: Retrieving Data Source Values

You can use the DB_LOOKUP function to retrieve a value from one data source when running a request against another data source, without joining or combining the two data sources.

DB_LOOKUP compares pairs of fields from the source and lookup data sources to locate matching records and retrieve the value to return to the request. You can specify as many pairs as needed to get to the lookup record that has the value you want to retrieve. If your field list pairs do not lead to a unique lookup record, the first matching lookup record retrieved is used.

DB_LOOKUP can be called in a DEFINE command, TABLE COMPUTE command, MODIFY COMPUTE command, or DataMigrator flow.

There are no restrictions on the source file. The lookup file can be any non-FOCUS data source that is supported as the cross referenced file in a cluster join. The lookup fields used to find the matching record are subject to the rules regarding cross-referenced join fields for the lookup data source. A fixed format sequential file can be the lookup file if it is sorted in the same order as the source file.

Syntax: How to Retrieve a Value From a Lookup Data Source

```
DB_LOOKUP(look_mf, srcfld1, lookfld1, srcfld2, lookfld2, ..., returnfld);
```

where:

`look_mf`

Is the lookup Master File.

srcfld1, srcfld2 ...

Are fields from the source file used to locate a matching record in the lookup file.

lookfld1, lookfld2 ...

Are columns from the lookup file that share values with the source fields. Only columns in the table or file can be used; columns created with DEFINE cannot be used. For multi-segment synonyms, only columns in the top segment can be used.

returnfld

Is the name of a column in the lookup file whose value is returned from the matching lookup record. Only columns in the table or file can be used; columns created with DEFINE cannot be used.

Reference: Usage Notes for DB_LOOKUP

- The maximum number of pairs that can be used to match records is 63.
- If the lookup file is a fixed format sequential file, it must be sorted and retrieved in the same order as the source file, unless the ENGINE INT SET CACHE=ON command is in effect. Having this setting in effect may also improve performance if the values will be looked up more than once. The key field of the sequential file must be the first lookup field specified in the DB_LOOKUP request. If it is not, no records will match.

  In addition, if a DB_LOOKUP request against a sequential file is issued in a DEFINE FILE command, you must clear the DEFINE FILE command at the end of the TABLE request that references it, or the lookup file will remain open. It will not be reusable until closed and may cause problems when you exit. Other types of lookup files can be reused without clearing the DEFINE. They will be cleared automatically when all DEFINE fields are cleared.
- If the lookup field has the MISSING=ON attribute in its Master File and the DEFINE or COMPUTE command specifies MISSING ON, the missing value is returned when the lookup field is missing. Without MISSING ON in both places, the missing value is converted to a default value (blank for an alphanumeric field, zero for a numeric field).
- Source records display on the report output even if they lack a matching record in the lookup file.
- Only real fields in the lookup Master File are valid as lookup and return fields.
- If there are multiple rows in the lookup table where the source field is equal to the lookup field, the first value of the return field is returned.

Example: Retrieving a Value From a LOOKUP Table

DB_LOOKUP takes the value for STORE_CODE and retrieves the STORENAME associated with it.

```
DB_LOOKUP(dmcomp,STORE_CODE,STORE_CODE,STORENAME)
```

For 1003CA the result is Audio Expert.

For 1004MD the result is City Video For 2010AZ the result is eMart.

DECODE: Decoding Values

The DECODE function assigns values based on the coded value of an input field. DECODE is useful for giving a more meaningful value to a coded value in a field. For example, the field GENDER may have the code F for female employees and M for male employees for efficient storage (for example, one character instead of six for *female*). DECODE expands (decodes) these values to ensure correct interpretation on a report.

You can use DECODE by supplying values directly in the function or by reading values from a separate file.

Syntax: How to Supply Values in the Function

```
DECODE fieldname(code1 result1 code2 result2...[ELSE default ]);
DECODE fieldname(filename ...[ELSE default]);
```

where:

fieldname

Alphanumeric or Numeric

Is the name of the input field.

code

Alphanumeric or Numeric

Is the coded value that DECODE compares with the current value of *fieldname*. If the value has embedded blanks, commas, or other special characters, it must be enclosed in single quotation marks. When DECODE finds the specified value, it returns the corresponding result. When the code is compared to the value of the field name, the code and field name must be in the same format.

result

Alphanumeric or Numeric

Is the returned value that corresponds to the code. If the result has embedded blanks or commas, or contains a negative number, it must be enclosed in single quotation marks. Do not use double quotation marks (").

If the result is presented in alphanumeric format, it must be a non-null, non-blank string. The format of the result must correspond to the data type of the expression.

default

Alphanumeric or Numeric

Is the value returned as a result for non-matching codes. The format must be the same as the format of *result*. If you omit a default value, DECODE assigns a blank or zero to non-matching codes.

filename

Alphanumeric

Is the name of the file in which code/result pairs are stored. Every record in the file must contain a pair.

You can use up to 40 lines to define the code and result pairs for any given DECODE function, or 39 lines if you also use an ELSE phrase. Use either a comma or blank to separate the code from the result, or one pair from another.

Note: DECODE has no *output* argument.

Example: Supplying Values Using the DECODE Function

DECODE returns the state abbreviation for PLANT.

```
DECODE PLANT(BOS 'MA' DAL 'TX' LA 'CA')
```

For BOS, the result is MA.

For DAL, the result is TX.

For LA, the result is CA.

FIND: Verifying the Existence of a Value in a Data Source

The FIND function determines if an incoming data value is in an indexed FOCUS data source field. The function sets a temporary field to a non-zero value if the incoming value is in the data source field, and to 0 if it is not. A value greater than zero confirms the presence of the data value, not the number of instances in the data source field.

You can also use FIND in a VALIDATE command to determine if a transaction field value exists in another FOCUS data source. If the field value is not in that data source, the function returns a value of 0, causing the validation test to fail and the request to reject the transaction.

You can use any number of FINDs in a COMPUTE or VALIDATE command. However, more FINDs increase processing time and require more buffer space in memory.

Limit: FIND does not work on files with different DBA passwords.

The opposite of FIND is NOT FIND. The NOT FIND function sets a temporary field to *1* if the incoming value is not in the data source and to *0* if the incoming value is in the data source.

Syntax: How to Verify the Existence of a Value in a Data Source

```
FIND(fieldname [AS dbfield] IN file);
```

where:

fieldname

Is the name of the field that contains the incoming data value.

AS *dbfield*

Is the name of the data source field whose values are compared to the incoming field values.

This field must be indexed. If the incoming field and the data source field have the same name, omit this phrase.

file

Is the name of the indexed FOCUS data source.

Note:

- FIND does not use an *output* argument.
- Do not include a space between FIND and the left parenthesis.

Example: Verifying the Existence of a Value in an Indexed Field

FIND determines if a supplied value in EMP_ID is in the EDUCFILE data source.

```
FIND(EMP_ID IN EDUCFILE)
```

IMPUTE: Replacing Missing Values With Aggregated Values

IMPUTE calculates a value to replace missing numeric data on report output, within a partition.

In place of eliminating data records with missing values from analysis, IMPUTE enables you to substitute a variety of estimates for the missing values, including the mean, the median, the mode, or a numeric constant, all calculated within the data partition specified by the reset key. This function is designed to be used with detail level reports (PRINT or LIST commands), and with calculated values (fields created with the COMPUTE command).

Syntax: How to Replace Missing Values With Aggregated Values

```
IMPUTE(field, reset_key, replacement)
```

where:

field

Is the name of the numeric input field that is defined with MISSING ON.

reset_key

Defines the partition for the calculation. Valid values are:

- A sort field name.
- PRESET, which uses the break defined by the SET PARTITION_ON command.
- TABLE, which performs the calculation on the entire table.

replacement

Is a numeric constant or one of the following:

- MEAN
- MEDIAN
- MODE

Example: Replacing Missing Values With Aggregated Values

To run this example, the FOCUS data source SALEMISS must be created. SALEMISS is the SALES data source with some missing values added in the RETURNS and DAMAGED fields. The following is the SALEMISS Master File, which should be added to the IBISAMP application.

```
FILENAME=KSALES, SUFFIX=FOC, REMARKS='Legacy Metadata Sample: sales',$

SEGNAME=STOR_SEG, SEGTYPE=S1,
   FIELDNAME=STORE_CODE,  ALIAS=SNO,  FORMAT=A3,   $
   FIELDNAME=CITY,        ALIAS=CTY,  FORMAT=A15,  $
   FIELDNAME=AREA,        ALIAS=LOC,  FORMAT=A1,   $

SEGNAME=DATE_SEG, PARENT=STOR_SEG, SEGTYPE=SH1,
   FIELDNAME=DATE,        ALIAS=DTE,  FORMAT=A4MD, $

SEGNAME=PRODUCT, PARENT=DATE_SEG, SEGTYPE=S1,
   FIELDNAME=PROD_CODE,     ALIAS=PCODE,   FORMAT=A3,    FIELDTYPE=I, $
   FIELDNAME=UNIT_SOLD,     ALIAS=SOLD,    FORMAT=I5,    $
   FIELDNAME=RETAIL_PRICE,  ALIAS=RP,      FORMAT=D5.2M, $
   FIELDNAME=DELIVER_AMT,   ALIAS=SHIP,    FORMAT=I5,    $
   FIELDNAME=OPENING_AMT,   ALIAS=INV,     FORMAT=I5,    $
   FIELDNAME=RETURNS,       ALIAS=RTN,     FORMAT=I3,    MISSING=ON, $
   FIELDNAME=DAMAGED,       ALIAS=BAD,     FORMAT=I3,    MISSING=ON, $
```

The following procedure creates the SALEMISS data source and then adds the missing values to the RETURNS and DAMAGED fields:

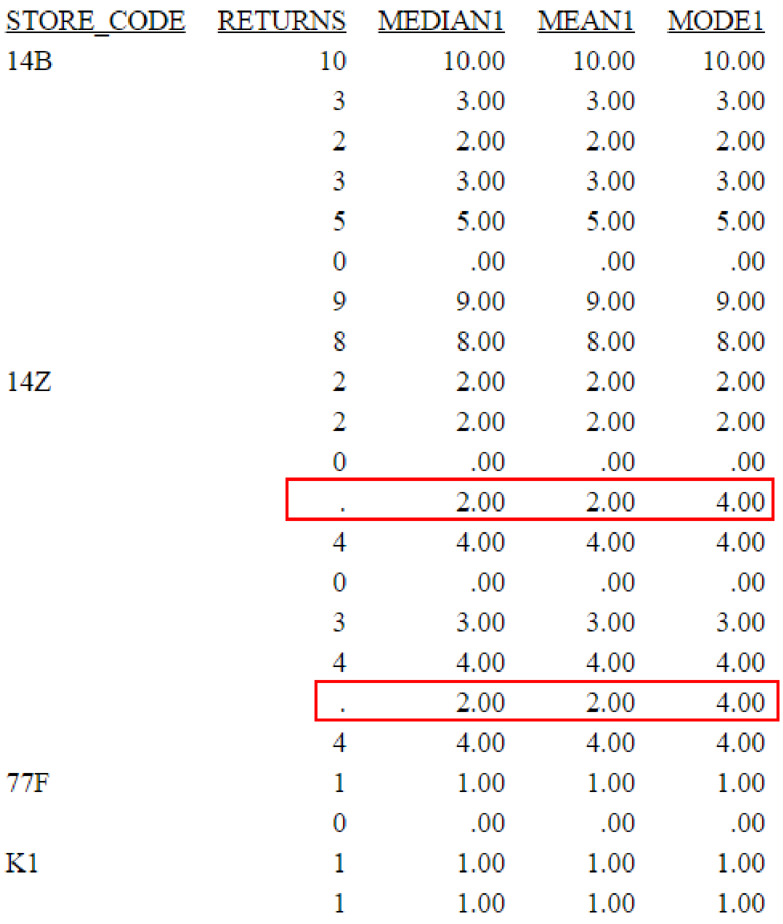
```
CREATE FILE ibisamp/SALEMISS
MODIFY FILE ibisamp/SALEMISS
FIXFORM STORE_CODE/3 CITY/15 AREA/1 DATE/4 PROD_CODE/3
FIXFORM UNIT_SOLD/5 RETAIL_PRICE/5 DELIVER_AMT/5
FIXFORM OPENING_AMT/5 RETURNS/3 DAMAGED/3
MATCH STORE_CODE
ON NOMATCH INCLUDE
ON MATCH CONTINUE
MATCH DATE
ON NOMATCH INCLUDE
ON MATCH CONTINUE
MATCH PROD_CODE
ON NOMATCH INCLUDE
ON MATCH REJECT
DATA
14BSTAMFORD       S1212B10   60  .95   80   65 10  6
14BSTAMFORD       S1212B12   40 1.29   20   50  3  3
14BSTAMFORD       S1212B17   29 1.89   30   30  2  1
14BSTAMFORD       S1212C13   25 1.99   30   40  3  0
14BSTAMFORD       S1212C7    45 2.39   50   49  5  4
14BSTAMFORD       S1212D12   27 2.19   40   35  0  0
14BSTAMFORD       S1212E2    80  .99  100  100  9  4
14BSTAMFORD       S1212E3    70 1.09   80   90  8  9
14ZNEW YORK       U1017B10   30  .85   30   10  2  3
14ZNEW YORK       U1017B17   20 1.89   40   25  2  1
14ZNEW YORK       U1017B20   15 1.99   30    5  0  1
14ZNEW YORK       U1017C17   12 2.09   10   15  0  0
14ZNEW YORK       U1017D12   20 2.09   30   10  3  2
14ZNEW YORK       U1017E1    30  .89   25   45  4  7
14ZNEW YORK       U1017E3    35 1.09   25   45  4  2
77FUNIONDALE      R1018B20   25 2.09   40   25  1  1
77FUNIONDALE      R1018C7    40 2.49   40   40  0  0
K1 NEWARK         U1019B12   29 1.49   30   30  1  0
K1 NEWARK         U1018B10   13  .99   30   15  1  1
END
-RUN
```

```
MODIFY FILE ibisamp/SALEMISS
FIXFORM STORE_CODE/3 DATE/5 PROD_CODE/4
FIXFORM UNIT/3 RETAIL/5 DELIVER/3
FIXFORM OPEN/3 RETURNS/C3 DAMAGED/C3
MATCH STORE_CODE
ON NOMATCH INCLUDE
ON MATCH CONTINUE
MATCH DATE
ON NOMATCH INCLUDE
ON MATCH CONTINUE
MATCH PROD_CODE
ON NOMATCH INCLUDE
ON MATCH REJECT
DATA
14Z1017 C13 15 1.99 35 30    6
14Z1017 C14 18 2.05 30 25 4
14Z1017 E2  33 0.99 45 40
END
-RUN
```

The following request against the SALEMISS data source generates replacement values for the missing values in the RETURNS field, using only the values within the same store.

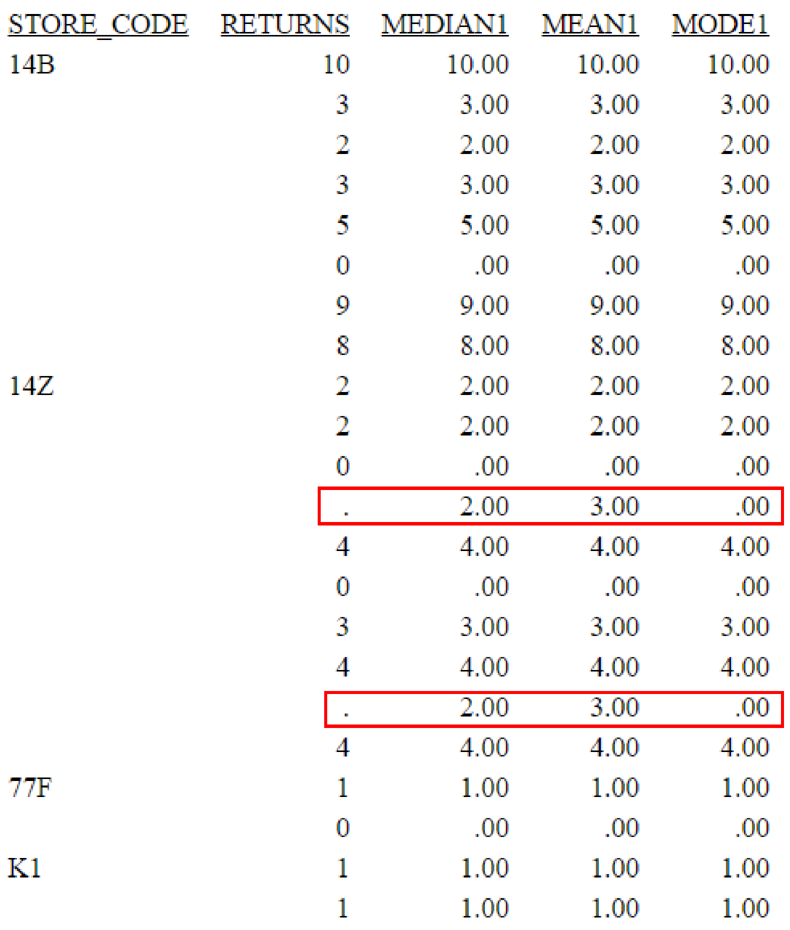
```
SET PARTITION_ON=FIRST
TABLE FILE SALEMISS
PRINT RETURNS
COMPUTE MEDIAN1 = IMPUTE(RETURNS, PRESET, MEDIAN);
COMPUTE MEAN1 = IMPUTE(RETURNS, PRESET, MEAN);
COMPUTE MODE1 = IMPUTE(RETURNS, PRESET, MODE);
BY STORE_CODE
ON TABLE SET PAGE NOPAGE
ON TABLE SET STYLE *
TYPE=REPORT, GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. The missing values occur in store 14Z, and the replacement values are calculated using only the RETURNS values from that store because PARTITION_ON is set to FIRST.



STORE_CODE	RETURNS	MEDIAN1	MEAN1	MODE1
14B	10	10.00	10.00	10.00
	3	3.00	3.00	3.00
	2	2.00	2.00	2.00
	3	3.00	3.00	3.00
	5	5.00	5.00	5.00
	0	.00	.00	.00
	9	9.00	9.00	9.00
	8	8.00	8.00	8.00
14Z	2	2.00	2.00	2.00
	2	2.00	2.00	2.00
	0	.00	.00	.00
	.	2.00	2.00	4.00
	4	4.00	4.00	4.00
	0	.00	.00	.00
	3	3.00	3.00	3.00
	4	4.00	4.00	4.00
	.	2.00	2.00	4.00
	4	4.00	4.00	4.00
77F	1	1.00	1.00	1.00
	0	.00	.00	.00
K1	1	1.00	1.00	1.00
	1	1.00	1.00	1.00

Changing the PARTITION_ON setting to TABLE produces the following output, in which the replacement values are calculated using all of the rows in the table.



```
STORE_CODE  RETURNS  MEDIAN1  MEAN1  MODE1
14B              10    10.00  10.00  10.00
                  3     3.00   3.00   3.00
                  2     2.00   2.00   2.00
                  3     3.00   3.00   3.00
                  5     5.00   5.00   5.00
                  0      .00    .00    .00
                  9     9.00   9.00   9.00
                  8     8.00   8.00   8.00
14Z               2     2.00   2.00   2.00
                  2     2.00   2.00   2.00
                  0      .00    .00    .00
                  .     2.00   3.00    .00
                  4     4.00   4.00   4.00
                  0      .00    .00    .00
                  3     3.00   3.00   3.00
                  4     4.00   4.00   4.00
                  .     2.00   3.00    .00
                  4     4.00   4.00   4.00
77F               1     1.00   1.00   1.00
                  0      .00    .00    .00
K1                1     1.00   1.00   1.00
                  1     1.00   1.00   1.00
```

LAST: Retrieving the Preceding Value

The LAST function retrieves the preceding value for a field.

The effect of LAST depends on whether it appears in an extract or load transformation:

- In an extract transformation the LAST value applies to the previous record retrieved from the data source before sorting takes place.

- ❏ In a load transformation, the LAST value applies to the record in the previous record loaded.

Syntax: How to Retrieve the Preceding Value

```
LAST fieldname
```

where:

fieldname

Alphanumeric or Numeric

Is the field name.

Note: LAST does not use an *output* argument.

Example: Retrieving the Preceding Value

LAST retrieves the previous value of DEPARTMENT:

```
LAST DEPARTMENT
```

LOOKUP: Retrieving a Value From a Cross-referenced Data Source

The LOOKUP function retrieves a data value from a cross-referenced FOCUS data source in a MODIFY request. You can retrieve data from a data source cross-referenced statically in a synonym or a data source joined dynamically to another by the JOIN command. LOOKUP retrieves a value, but does not activate the field. LOOKUP is required because a MODIFY request, unlike a TABLE request, cannot read cross-referenced data sources freely.

LOOKUP allows a request to use the retrieved data in a computation or message, but it does not allow you to modify a cross-referenced data source.

LOOKUP can read a cross-referenced segment that is linked directly to a segment in the host data source (the host segment). This means that the cross-referenced segment must have a segment type of KU, KM, DKU, or DKM (but not KL or KLU) or must contain the cross-referenced field specified by the JOIN command. Because LOOKUP retrieves a single cross-referenced value, it is best used with unique cross-referenced segments.

The cross-referenced segment contains two fields used by LOOKUP:

- ❏ The field containing the retrieved value. Alternatively, you can retrieve all the fields in a segment at one time. The field, or your decision to retrieve all the fields, is specified in LOOKUP.

  For example, LOOKUP retrieves all the fields from the segment

```
RTN = LOOKUP(SEG.DATE_ATTEND);
```

- The cross-referenced field. This field shares values with a field in the host segment called the host field. These two fields link the host segment to the cross-referenced segment. LOOKUP uses the cross-referenced field, which is indexed, to locate a specific segment instance.

When using LOOKUP, the MODIFY request reads a transaction value for the host field. It then searches the cross-referenced segment for an instance containing this value in the cross-referenced field:

- If there are no instances of the value, the function sets a return variable to 0. If you use the field specified by LOOKUP in the request, the field assumes a value of blank if alphanumeric and 0 if numeric.
- If there are instances of the value, the function sets the return variable to 1 and retrieves the value of the specified field from the first instance it finds. There can be more than one if the cross-referenced segment type is KM or DKM, or if you specified the ALL keyword in the JOIN command.

Syntax: How to Retrieve a Value From a Cross-referenced Data Source

```
LOOKUP(field);
```

where:

field

Is the name of the field to retrieve in the cross-referenced file. If the field name also exists in the host data source, you must qualify it here. Do not include a space between LOOKUP and the left parenthesis.

Note: LOOKUP does not use an *output* argument.

Example: Using the LOOKUP Function

LOOKUP finds the enrollment date from DATE_ENROLL. The result can then be used to validate an expression.

```
LOOKUP(DATE_ENROLL)
```

NULLIF: Returning a Null Value When Parameters Are Equal

NULLIF returns a null (missing) value when its parameters are equal. If they are not equal, it returns the first value. The field to which the value is returned should have MISSING ON.

Syntax: How to Return a Null Value for Equal Parameters

```
NULLIF(arg1,arg2)
```

where:

arg1,arg2

Any type of field, constant, or expression.

Are the input parameters that are tested for equality. They must either both be numeric or both be alphanumeric.

The output data type is the same as the input data types.

Example: Testing for Equal Parameters

NULLIF tests the DAMAGED and RETURNS field values for equality.

```
NULLIF(DAMAGED, RETURNS)
```

For DAMAGED=3 and RETURNS = 3, the result is MISSING (.).

For DAMAGED=2 and RETURNS = 3, the result is 2.

Chapter 8

Simplified Date and Date-Time Functions

Simplified date and date-time functions have streamlined parameter lists, similar to those used by SQL functions. In some cases, these simplified functions provide slightly different functionality than previous versions of similar functions.

The simplified functions do not have an output argument. Each function returns a value that has a specific data type.

When used in a request against a relational data source, these functions are optimized (passed to the RDBMS for processing).

Standard date and date-time formats refer to YYMD and HYYMD syntax (dates that are not stored in alphanumeric or numeric fields). Dates not in these formats must be converted before they can be used in the simplified functions. Input date and date-time parameters must provide full component dates. Literal date-time values can be used with the DT function.

All arguments can be either literals, field names, or amper variables.

In this chapter:

- MONTHNAME: Returning the Name of the Month From a Date Expression

DAYNAME: Returning the Name of the Day From a Date Expression

DAYNAME returns a character string that contains the data-source-specific name of the day for the day part of a date expression.

Syntax: How to Return the Name of the Day From a Date Expression

```
DAYNAME(date_exp)
```

where:

date_exp
: Is a date or date-time expression.

Example: Returning the Name of the Day From a Date Expression

DAYNAME returns the name of the day.

```
DAYNAME(TIME_DATE)
```

For January 1, 2009, the result is Thursday.

DT_CURRENT_DATE: Returning the Current Date

The DT_CURRENT_DATE function returns the current date-time provided by the running operating environment in date-time format. The time portion of the date-time is set to zero.

Syntax: How to Return the Current Date

```
DT_CURRENT_DATE()
```

Example: Returning the Current Date

DT_CURRENT_DATE returns the current date.

```
DT_CURRENT_DATE()
```

For September 8, 2016 (returning to a YYMD field), the result is 2016/09/08.

DT_CURRENT_DATETIME: Returning the Current Date and Time

DT_CURRENT_DATETIME returns the current date and time provided by the running operating environment in date-time format, with a specified time precision.

Syntax: How to Return the Current Date and Time

```
DT_CURRENT_DATETIME(component)
```

where:

`component`

Is one of the following time precisions.

- SECOND.
- MILLISECOND.
- MICROSECOND.

Note: The field to which the value is returned must have a format that supports the time precision requested.

Example: Returning the Current Date and Time

DT_CURRENT_DATETIME returns the current date and time to microsecond precision.

```
DT_CURRENT_DATETIME(MICROSECOND)
```

For September 8,2106 at 5:10:31.605718 p.m. (returned to a field with format HYYMDm), the result is 2016/09/08 17:10:31.605718.

DT_CURRENT_TIME: Returning the Current Time

The DT_CURRENT_TIME function returns the current time provided by the running operating environment in date-time format, with a specified time precision. The date portion of the returned date-time value is set to zero.

Syntax: How to Return the Current Time

```
DT_CURRENT_TIME(component)
```

where:

`component`

Is one of the following time precisions.

- SECOND.
- MILLISECOND.
- MICROSECOND.

Note: The field to which the value is returned must have a format that supports the time precision requested.

Example: **Returning the Current Time**

DT_CURRENT_TIME returns the current time in milliseconds.

```
DT_CURRENT_TIME(MILLISECOND)
```

For 5:23:13.098 p.m. (returned to a field with format HHISs), the result is 17:23:13.098.

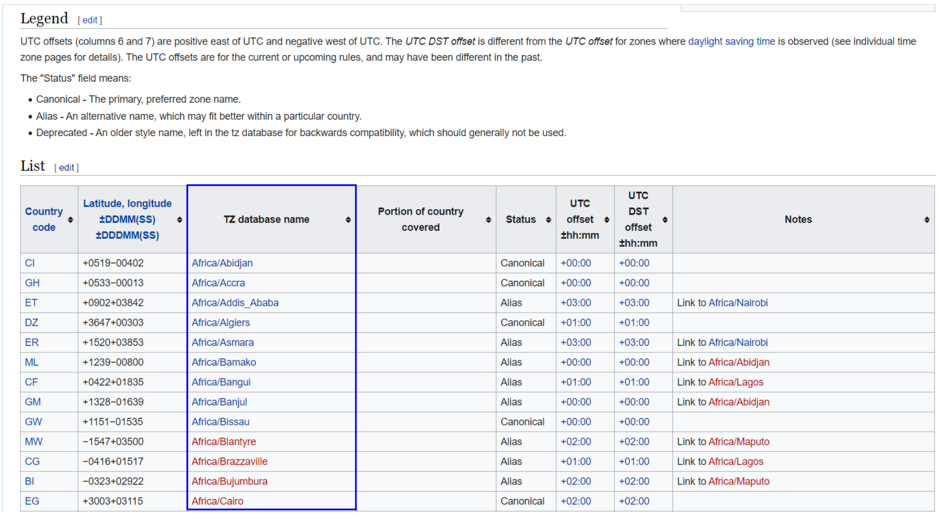
DT_TOLOCAL: Converting Universal Coordinated Time to Local Time

Coordinated Universal Time (UTC) is the time standard commonly used around the world. To convert UTC time to a local time, a certain number of hours must be added to or subtracted from the UTC time, depending on the number of time zones between the locality and Greenwich, England (GMT).

DT_TOLOCAL converts UTC time to local time.

Converting timestamp values from different localities to a common standard time enables you to sort events into the actual event sequence.

This function requires an IANA (Internet Assigned Numbers Authority) time zone database names (expressed as 'Area/Location') as a parameter. You can find a table of IANA TZ database names on Wikipedia at *https://en.wikipedia.org/wiki/List_of_tz_database_time_zones*, as shown in the following image.



Legend [edit]

UTC offsets (columns 6 and 7) are positive east of UTC and negative west of UTC. The *UTC DST offset* is different from the *UTC offset* for zones where daylight saving time is observed (see individual time zone pages for details). The UTC offsets are for the current or upcoming rules, and may have been different in the past.

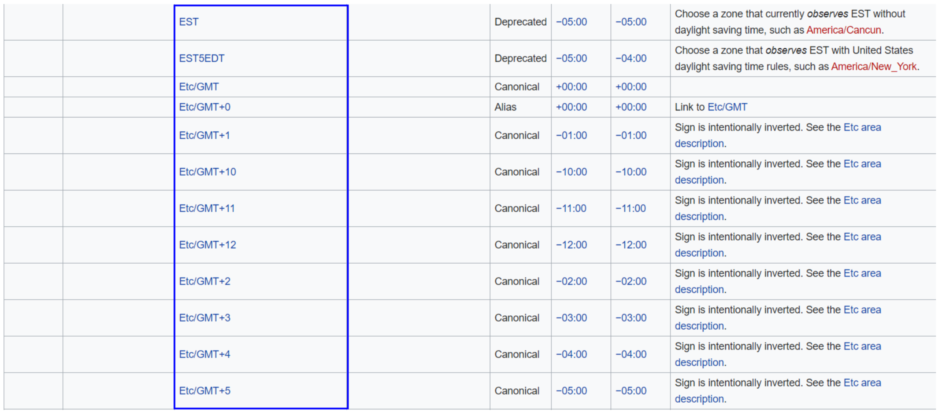
The "Status" field means:

- Canonical - The primary, preferred zone name.
- Alias - An alternative name, which may fit better within a particular country.
- Deprecated - An older style name, left in the tz database for backwards compatibility, which should generally not be used.

List [edit]

Country code	Latitude, longitude ±DDMM(SS) ±DDDMM(SS)	TZ database name	Portion of country covered	Status	UTC offset ±hh:mm	UTC DST offset ±hh:mm	Notes
CI	+0519−00402	Africa/Abidjan		Canonical	+00:00	+00:00	
GH	+0533−00013	Africa/Accra		Canonical	+00:00	+00:00	
ET	+0902+03842	Africa/Addis_Ababa		Alias	+03:00	+03:00	Link to Africa/Nairobi
DZ	+3647+00303	Africa/Algiers		Canonical	+01:00	+01:00	
ER	+1520+03853	Africa/Asmara		Alias	+03:00	+03:00	Link to Africa/Nairobi
ML	+1239−00800	Africa/Bamako		Alias	+00:00	+00:00	Link to Africa/Abidjan
CF	+0422+01835	Africa/Bangui		Alias	+01:00	+01:00	Link to Africa/Lagos
GM	+1328−01639	Africa/Banjul		Alias	+00:00	+00:00	Link to Africa/Abidjan
GW	+1151−01535	Africa/Bissau		Canonical	+00:00	+00:00	
MW	−1547+03500	Africa/Blantyre		Alias	+02:00	+02:00	Link to Africa/Maputo
CG	−0416+01517	Africa/Brazzaville		Alias	+01:00	+01:00	Link to Africa/Lagos
BI	−0323+02922	Africa/Bujumbura		Alias	+02:00	+02:00	Link to Africa/Maputo
EG	+3003+03115	Africa/Cairo		Canonical	+02:00	+02:00	

If you do not know what Area and Location corresponds to your time zone, but you do know your offset from GMT, or your legacy time zone name (such as EST), scroll down in the table. There are TZ database names that correspond to these time zone identifiers, as shown in the following image.



		EST		Deprecated	−05:00	−05:00	Choose a zone that currently *observes* EST without daylight saving time, such as America/Cancun.
		EST5EDT		Deprecated	−05:00	−04:00	Choose a zone that *observes* EST with United States daylight saving time rules, such as America/New_York.
		Etc/GMT		Canonical	+00:00	+00:00	
		Etc/GMT+0		Alias	+00:00	+00:00	Link to Etc/GMT
		Etc/GMT+1		Canonical	−01:00	−01:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+10		Canonical	−10:00	−10:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+11		Canonical	−11:00	−11:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+12		Canonical	−12:00	−12:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+2		Canonical	−02:00	−02:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+3		Canonical	−03:00	−03:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+4		Canonical	−04:00	−04:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+5		Canonical	−05:00	−05:00	Sign is intentionally inverted. See the Etc area description.

Note: If you use a standard IANA time zone database name in the form 'Area/Location' (for example, 'America/New_York'), automatic adjustments are made for Daylight Savings Time. If you use a name that corresponds to an offset from GMT or to a legacy time zone name, it is your responsibility to account for Daylight Savings Time.

Syntax:

How to Convert UTC Time to Local Time

```
DT_TOLOCAL(datetime, timezone)
```

where:

datetime

Date-time

Is a date-time expression representing UTC time, containing date and time components.

timezone

Alphanumeric

Is a character expression containing the IANA time zone name of the local time, in the form 'Area/Location' (for example, 'America/New_York').

Example:

Converting UTC Time to Local Time

DT_TOLOCAL converts the column UTC1 to local time in time zone 'America/New_York'.

```
DT_TOLOCAL(UTC1, 'America/New_York')
```

For '2020/09/04 15:00:26', the result is '2020/09/04 11:00:26'.

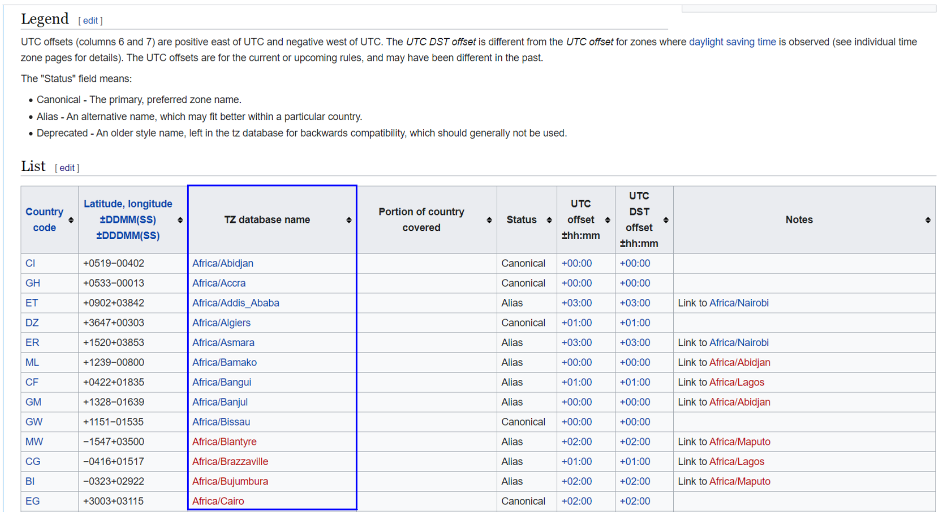
DT_TOUTC: Converting Local Time to Universal Coordinated Time

Coordinated Universal Time (UTC) is the time standard commonly used around the world. To convert UTC time to a local time, a certain number of hours must be added to or subtracted from the UTC time, depending on the number of time zones between the locality and Greenwich, England (GMT).

DT_TOUTC converts local time to UTC time.

Converting timestamp values from different localities to a common standard time enables you to sort events into the actual event sequence.

This function requires an IANA (Internet Assigned Numbers Authority) time zone database names (expressed as 'Area/Location') as a parameter. You can find a table of IANA TZ database names on Wikipedia at *https://en.wikipedia.org/wiki/List_of_tz_database_time_zones*, as shown in the following image.



Legend [edit]

UTC offsets (columns 6 and 7) are positive east of UTC and negative west of UTC. The *UTC DST offset* is different from the *UTC offset* for zones where daylight saving time is observed (see individual time zone pages for details). The UTC offsets are for the current or upcoming rules, and may have been different in the past.

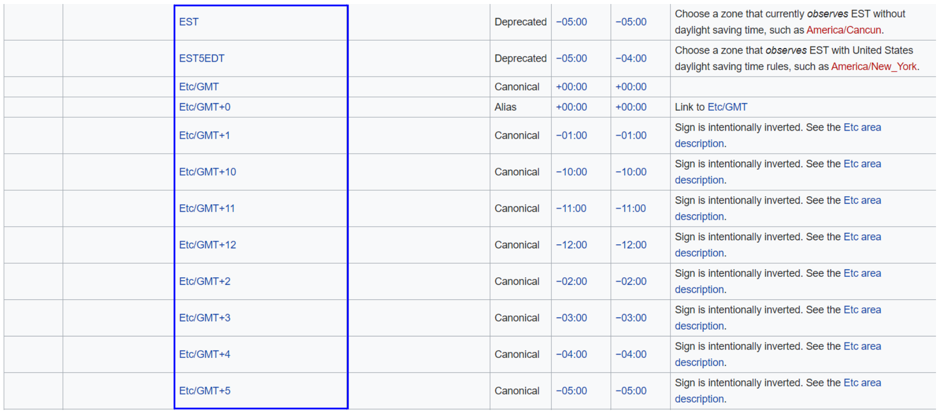
The "Status" field means:

- Canonical - The primary, preferred zone name.
- Alias - An alternative name, which may fit better within a particular country.
- Deprecated - An older style name, left in the tz database for backwards compatibility, which should generally not be used.

List [edit]

Country code	Latitude, longitude ±DDMM(SS) ±DDDMM(SS)	TZ database name	Portion of country covered	Status	UTC offset ±hh:mm	UTC DST offset ±hh:mm	Notes
CI	+0519−00402	Africa/Abidjan		Canonical	+00:00	+00:00	
GH	+0533−00013	Africa/Accra		Canonical	+00:00	+00:00	
ET	+0902+03842	Africa/Addis_Ababa		Alias	+03:00	+03:00	Link to Africa/Nairobi
DZ	+3647+00303	Africa/Algiers		Canonical	+01:00	+01:00	
ER	+1520+03853	Africa/Asmara		Alias	+03:00	+03:00	Link to Africa/Nairobi
ML	+1239−00800	Africa/Bamako		Alias	+00:00	+00:00	Link to Africa/Abidjan
CF	+0422+01835	Africa/Bangui		Alias	+01:00	+01:00	Link to Africa/Lagos
GM	+1328−01639	Africa/Banjul		Alias	+00:00	+00:00	Link to Africa/Abidjan
GW	+1151−01535	Africa/Bissau		Canonical	+00:00	+00:00	
MW	−1547+03500	Africa/Blantyre		Alias	+02:00	+02:00	Link to Africa/Maputo
CG	−0416+01517	Africa/Brazzaville		Alias	+01:00	+01:00	Link to Africa/Lagos
BI	−0323+02922	Africa/Bujumbura		Alias	+02:00	+02:00	Link to Africa/Maputo
EG	+3003+03115	Africa/Cairo		Canonical	+02:00	+02:00	

If you do not know what Area and Location corresponds to your time zone, but you do know your offset from GMT, or your legacy time zone name (such as EST), scroll down in the table. There are TZ database names that correspond to these time zone identifiers, as shown in the following image.



		EST		Deprecated	−05:00	−05:00	Choose a zone that currently *observes* EST without daylight saving time, such as America/Cancun.
		EST5EDT		Deprecated	−05:00	−04:00	Choose a zone that *observes* EST with United States daylight saving time rules, such as America/New_York.
		Etc/GMT		Canonical	+00:00	+00:00	
		Etc/GMT+0		Alias	+00:00	+00:00	Link to Etc/GMT
		Etc/GMT+1		Canonical	−01:00	−01:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+10		Canonical	−10:00	−10:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+11		Canonical	−11:00	−11:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+12		Canonical	−12:00	−12:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+2		Canonical	−02:00	−02:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+3		Canonical	−03:00	−03:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+4		Canonical	−04:00	−04:00	Sign is intentionally inverted. See the Etc area description.
		Etc/GMT+5		Canonical	−05:00	−05:00	Sign is intentionally inverted. See the Etc area description.

Note: If you use a standard IANA time zone database name in the form 'Area/Location' (for example, 'America/New_York'), automatic adjustments are made for Daylight Savings Time. If you use a name that corresponds to an offset from GMT or to a legacy time zone name, it is your responsibility to account for Daylight Savings Time.

Syntax: How to Convert Local Time to UTC Time

```
DT_TOUTC(datetime, timezone)
```

where:

datetime

Date-time

Is a date-time expression representing local time, containing date and time components.

timezone

Alphanumeric

Is a character expression containing the IANA time zone name of the local time, in the form 'Area/Location' (for example, 'America/New_York').

Example: Converting Local Time to UTC Time

DT_TOUTC converts the date-time column LOCAL1 in time zone 'America/New_York' to UTC time.

```
DT_TOUTC(LOCAL1, 'America/New_York')
```

For '2020/09/04 14:49:41', the result is '2020/09/04 18:49:41'.

DTADD: Incrementing a Date or Date-Time Component

Given a date in standard date or date-time format, DTADD returns a new date after adding the specified number of a supported component. The returned date format is the same as the input date format.

Syntax: How to Increment a Date or Date-Time Component

```
DTADD(date, component, increment)
```

where:

date

Date or date-time

Is the date or date-time value to be incremented, which must provide a full component date.

component

Keyword

Is the component to be incremented. Valid components (and acceptable values) are:

- YEAR (1-9999).
- QUARTER (1-4).
- MONTH (1-12).
- WEEK (1-53). This is affected by the WEEKFIRST setting.
- DAY (of the Month, 1-31).
- HOUR (0-23).
- MINUTE (0-59).
- SECOND (0-59).

increment

Integer

Is the value (positive or negative) to add to the component.

Example: Incrementing the DAY Component of a Date

DTADD adds three days to the employee date of birth:

```
DTADD(DATE_OF_BIRTH, DAY, 3)
```

For 1976/10/21, the result is 1976/10/24.

Reference: Usage Notes for DTADD

- Each element must be manipulated separately. Therefore, if you want to add 1 year and 1 day to a date, you need to call the function twice, once for YEAR (you need to take care of leap years) and once for DAY. The simplified functions can be nested in a single expression, or created and applied in separate DEFINE or COMPUTE expressions.
- With respect to parameter validation, DTADD will not allow anything but a standard date or a date-time value to be used in the first parameter.
- The increment is not checked, and the user should be aware that decimal numbers are not supported and will be truncated. Any combination of values that increases the YEAR beyond 9999 returns the input date as the value, with no message. If the user receives the input date when expecting something else, it is possible there was an error.

DTDIFF: Returning the Number of Component Boundaries Between Date or Date-Time Values

Given two dates in standard date or date-time formats, DTIFF returns the number of given component boundaries between the two dates. The returned value has integer format for calendar components or double precision floating point format for time components.

Syntax: How to Return the Number of Component Boundaries

```
DTDIFF(end_date, start_date, component)
```

where:

end_date

Date or date-time

Is the ending full-component date in either standard date or date-time format. If this date is given in standard date format, all time components are assumed to be zero.

`start_date`

Date or date-time

Is the starting full-component date in either standard date or date-time format. If this date is given in standard date format, all time components are assumed to be zero.

`component`

Keyword

Is the component on which the number of boundaries is to be calculated. For example, QUARTER finds the difference in quarters between two dates. Valid components (and acceptable values) are:

- ❑ YEAR (1-9999).
- ❑ QUARTER (1-4).
- ❑ MONTH (1-12).
- ❑ WEEK (1-53). This is affected by the WEEKFIRST setting.
- ❑ DAY (of the Month, 1-31).
- ❑ HOUR (0-23).
- ❑ MINUTE (0-59).
- ❑ SECOND (0-59).

Example: **Returning the Number of Years Between Two Dates**

DTDIFF calculates employee age when hired:

```
DTDIFF(START_DATE, DATE_OF_BIRTH, YEAR)
```

For the date of birth 1991/06/04 and the start date 2008/11/14, the result is 17.

DTDIFF calculates the difference between two date-time values in minutes:

```
DTDIFF(DATETIME1, DATETIME2, MINUTES)
```

For DATETIME1 = 2020/0116 12:25 and DATETIME2 = 2020/0116 12:20, the result is 5.

For DATETIME1 = 2020/0116 12:25 and DATETIME2 = 2020/0115 12:20, the result is 1445.

DTIME: Extracting Time Components From a Date-Time Value

Given a date-time value and time component keyword as input, DTIME returns the value of all of the time components up to and including the requested component. The remaining time components in the value are set to zero. The field to which the time component is returned must have a time format that supports the component being returned.

Syntax: How to Extract a Time Component From a Date-Time Value

```
DTIME(datetime, component)
```

where:

datetime

Date-time

Is the date-time value from which to extract the time component. It can be a field name or a date-time literal. It must provide a full component date.

component

Keyword

Valid values are:

- TIME. The complete time portion is returned. Its smallest component depends on the input date-time format. Nanoseconds are not supported or returned.
- HOUR. The time component up to and including the hour component is extracted.
- MINUTE. The time component up to and including the minute component is extracted.
- SECOND. The time component up to and including the second component is extracted.
- MILLISECOND. The time component up to and including the millisecond component is extracted.
- MICROSECOND. The time component up to and including the microsecond component is extracted.

Example: Extracting Time Components

DTIME extracts the TIME component from the data-time value 2018/01/17 05:45:22.777888.

```
DTIME(DT(2018/01/17 05:45:22.777888), TIME)
```

The result is 05:45:22.777888.

DTPART: Returning a Date or Date-Time Component in Integer Format

Given a date in standard date or date-time format and a component, DTPART returns the component value in integer format.

Syntax: How to Return a Date or Date-Time Component in Integer Format

```
DTPART(date, component)
```

where:

date

Date or date-time

Is the full-component date in standard date or date-time format.

component

Keyword

Is the component to extract in integer format. Valid components (and values) are:

- ❏ YEAR (1-9999).
- ❏ QUARTER (1-4).
- ❏ MONTH (1-12).
- ❏ WEEK (of the year, 1-53). This is affected by the WEEKFIRST setting.
- ❏ DAY (of the Month, 1-31).
- ❏ DAY_OF_YEAR (1-366).
- ❏ WEEKDAY (day of the week, 1-7). This is affected by the WEEKFIRST setting.
- ❏ HOUR (0-23).
- ❏ MINUTE (0-59).
- ❏ SECOND (0-59).
- ❏ MILLISECOND (0-999).
- ❏ MICROSECOND (0-999999).

Example: Extracting the Quarter Component as an Integer

DTPART extracts the quarter from the employee start date:

```
DTPART(START_DATE, QUARTER)
```

For 2009/04/11, the result is 2.

DTRUNC: Returning the Start of a Date Period for a Given Date

Given a date or timestamp and a component, DTRUNC returns the first date within the period specified by that component.

Syntax: How to Return the First or Last Date of a Date Period

```
DTRUNC(date_or_timestamp, date_period)
```

where:

date_or_timestamp

Date or date-time

Is the date or timestamp of interest, which must provide a full component date.

date_period

Is the period whose starting or ending date you want to find. Can be one of the following:

- DAY, returns the date that represents the input date (truncates the time portion, if there is one).
- YEAR, returns the date of the first day of the year.
- MONTH, returns the date of the first day of the month.
- QUARTER, returns the date of the first day in the quarter.
- WEEK, returns the date that represents the first date of the given week.

  By default, the first day of the week will be Sunday, but this can be changed using the WEEKFIRST parameter.
- YEAR_END, returns the last date of the year.
- QUARTER_END, returns the last date of the quarter.
- MONTH_END, returns the last date of the month.
- WEEK_END, returns the last date of the week.

Example: Returning the First Date in a Date Period

DTRUNC returns the first date of the quarter given the date of birth:

```
DTRUNC(DATE_OF_BIRTH,QUARTER)
```

For 1993/03/27, the result is 1993/03/01.

Example: Using the Start of Week Parameter for DTRUNC

DTRUNC returns the date that represents the start of the week.

```
DTRUNC(START_DATE, WEEK)
```

For 2013/01/15, the result is 2013/01/13

Example: Returning the Date of the Last Day of a Week

DTRUNC calculates the date of the end of the week.

```
WEEKEND/YYMD = DTRUNC(START_DATE, WEEK_END)
```

For 2013/01/15, the result is 2013/01/19.

MONTHNAME: Returning the Name of the Month From a Date Expression

MONTHNAME returns a character string that contains the data-source-specific name of the month for the month part of a date expression.

Syntax: How to Return the Name of the Month From a Date Expression

```
MONTHNAME(date_exp)
```

where:

date_exp

Is a date or date-time expression.

Example: Returning the Name of the Month From a Date Expression

MONTHNAME returns the name of the month.

```
MONTHNAME(DATE)
```

For 'August 3, 2020', the result is August.

Chapter 9

Date Functions

Date functions manipulate date values. There are two types of date functions:

- ❑ Standard date functions for use with non-legacy dates.
- ❑ Legacy date functions for use with legacy dates.

  If a date is in an alphanumeric or numeric field that contains date display options (for example, I6YMD), you must use the legacy date functions.

In this chapter:

Overview of Date Functions

The following explains the difference between the types of date functions:

- **Standard date** functions are for use with standard date formats, or just date formats. A date format refers to internally stored data that is capable of holding date components, such as century, year, quarter, month, and day. It does not include time components. A synonym does not specify an internal data type or length for a date format. Instead, it specifies display date components, such as D (day), M (month), Q (quarter), Y (2-digit year), or YY (4-digit year). For example, format MDYY is a date format that has three date components; it can be used in the USAGE attribute of a synonym. A real date value, such as March 9, 2004, described by this format is displayed as 03/09/2004, by default. Date formats can be full component and non-full component. Full component formats include all three letters, for example, D, M, and Y. JUL for Julian can also be included. All other date formats are non-full component. Some date functions require full component arguments for date fields, while others will accept full or non-full components. A date format was formerly called a smart date.

- **Legacy date** functions are for use with legacy dates only. A legacy date refers to formats with date edit options, such as I6YMD, A6MDY, I8YYMD, or A8MDYY. For example, A6MDY is a 6-byte alphanumeric string. The suffix MDY indicates the order in which the date components are stored in the field, and the prefix I or A indicates a numeric or alphanumeric form of representation. For example, a value '030599' can be assigned to a field with format A6MDY, which will be displayed as 03/05/99.

Date formats have an internal representation matching either numeric or alphanumeric format. For example, A6MDY matches alphanumeric format, YYMD and I6DMY match numeric format. When function output is a date in specified by *output*, it can be used either for assignment to another date field of this format, or it can be used for further data manipulation in the expression with data of matching formats. Assignment to another field of a different date format, will yield a random result.

All but three date functions deal with only one date format. The exceptions are DATECVT, HCNVRT, and HDATE, which convert one date type into another.

Using Standard Date Functions

When using standard date functions, you need to understand the settings that alter the behavior of these functions, as well as the acceptable formats and how to supply values in these formats.

You can affect the behavior of date functions in the following ways:

- ❏ Defining which days of the week are work days and which are not. Then, when you use a date function involving work days, dates that are not work days are ignored. For details, see *Specifying Work Days* on page 209.
- ❏ Determining whether to display leading zeros when a date function in Dialogue Manager returns a date. For details, see *Enabling Leading Zeros For Date and Time Functions in Dialogue Manager* on page 214.

For detailed information on each standard date function, see:

DATEADD: Adding or Subtracting a Date Unit to or From a Date on page 215

DATECVT: Converting the Format of a Date on page 217

DATEDIF: Finding the Difference Between Two Dates on page 219

DATEMOV: Moving a Date to a Significant Point on page 221

DATETRAN: Formatting Dates in International Formats on page 226

FIYR: Obtaining the Financial Year on page 242

FIQTR: Obtaining the Financial Quarter on page 244

FIYYQ: Converting a Calendar Date to a Financial Date on page 246

TODAY: Returning the Current Date on page 247

Specifying Work Days

You can determine which days are work days and which are not. Work days affect the DATEADD, DATEDIF, and DATEMOV functions. You identify work days as business days or holidays.

Specifying Business Days

Business days are traditionally Monday through Friday, but not every business has this schedule. For example, if your company does business on Sunday, Tuesday, Wednesday, Friday, and Saturday, you can tailor business day units to reflect that schedule.

Syntax: How to Set Business Days

```
SET BUSDAYS = smtwtfs
```

where:

smtwtfs

Is the seven character list of days that represents your business week. The list has a position for each day from Sunday to Saturday:

- ❏ To identify a day of the week as a business day, enter the first letter of that day in that day's position.
- ❏ To identify a non-business day, enter an underscore (_) in that day's position.

If a letter is not in its correct position, or if you replace a letter with a character other than an underscore, you receive an error message.

Example: Setting Business Days to Reflect Your Work Week

The following designates work days as Sunday, Tuesday, Wednesday, Friday, and Saturday:

```
SET BUSDAYS = S_TW_FS
```

Syntax: How to View the Current Setting of Business Days

```
? SET BUSDAYS
```

Specifying Holidays

You can specify a list of dates that are designated as holidays in your company. These dates are excluded when using functions that perform calculations based on working days. For example, if Thursday in a given week is designated as a holiday, the next working day after Wednesday is Friday.

To define a list of holidays, you must:

1. Create a holiday file using a standard text editor.
2. Select the holiday file by issuing the SET command with the HDAY parameter.

Reference: Rules for Creating a Holiday File

- ❏ Dates must be in YYMD format.
- ❏ Dates must be in ascending order.

- ❏ Each date must be on its own line.
- ❏ Each year for which data exists must be included or the holiday file is considered invalid. Calling a date function with a date value outside the range of the holiday file returns a zero for business day requests.

  If you are subtracting two dates in 2005, and the latest date in the holiday file is 20041231, the subtraction will not be performed. One way to avoid invalidating the holiday file is to put a date very far in the future in any holiday file you create (for example, 29991231), and then it will always be considered valid.
- ❏ You may include an optional description of the holiday, separated from the date by a space.

By default, the holiday file has a file name of the form HDAY*xxxx*.err and is on your path, or on z/OS under PDS deployment, is a member named HDAY*xxxx* of a PDS allocated to DDNAME ERRORS. In your procedure or request, you must issue the SET HDAY=*xxxx* command to identify the file or member name. Alternatively, you can define the file to have any name and be stored anywhere or, on z/OS under PDS deployment, allocate the holiday file as a sequential file of any name or as member HDAY*xxxx* of any PDS. For information about using non-default holiday file names, see *How to FILEDEF or DYNAM the Holiday File* on page 212.

Procedure: How to Create a Holiday File

1. In a text editor, create a list of dates designated as holidays using the *Rules for Creating a Holiday File* on page 210.
2. Save the file.

   If you are not using the default naming convention, see *How to FILEDEF or DYNAM the Holiday File* on page 212. If you are using the default naming convention, use the following instructions:

   In Windows and UNIX: The file must be HDAY*xxxx*.ERR

   In z/OS: The file must be a member of ERRORS named HDAY*xxxx*.

   where:

   xxxx

   Is a string of text four characters long.

Syntax: How to Select a Holiday File

```
SET HDAY = xxxx
```

where:

xxxx

Is the part of the name of the holiday file after HDAY. This string must be four characters long.

Example: Creating and Selecting a Holiday File

The following is the HDAYTEST file, which establishes holidays:

```
19910325 TEST HOLIDAY
19911225 CHRISTMAS
```

The following sets HDAYTEST as the holiday file:

```
SET BUSDAYS = SMTWTFS
SET HDAY = TEST
```

Syntax: How to FILEDEF or DYNAM the Holiday File

In all environments except z/OS under PDS deployment, use the following syntax.

```
FILEDEF HDAYxxxx DISK {app/|path}/filename.ext
```

where:

HDAY*xxxx*

Is the logical name (DDNAME) for the holiday file, where *xxxx* is any four characters. You establish this logical name by issuing the SET HDAY=*xxxx* command in your procedure or request.

app

Is the name of the application in which the holiday file resides.

path

Is the path to the holiday file.

filename.ext

Is the name of the holiday file.

On z/OS under PDS deployment, use the following to allocate a sequential holiday file.

```
DYNAM ALLOC {DD|FILE} HDAYxxxx DA qualif.filename.suffix SHR REU
```

On z/OS under PDS deployment, use the following to allocate a holiday file that is a member of a PDS.

```
DYNAM ALLOC {DD|FILE} HDAYxxxx DA qualif.filename.suffix(HDAYxxx) SHR REU
```

where:

`HDAYxxxx`

Is the DDNAME for the holiday file. Your FOCEXEC or request must set the HDAY parameter to *xxxx*, where *xxxx* is any four characters you choose. If your holiday file is a member of a PDS, HDAY*xxxx* must also be the member name.

`qualif.filename.suffix`

Is the fully-qualified name of the sequential file that contains the list of holidays or the PDS with member HDAY*xxxx* that contains the list of holidays.

Example: Defining a Holiday File

The following holiday file, named holiday.data in the c:\temp directory on Windows, defines November 3, 2011 and December 24, 2011 as holidays:

```
20111103
20111224
```

The following defines and sets the holiday file. Then DATEADD finds the next business day taking the holiday file into account:

```
FILEDEF HDAYMMMM DISK c:\ibi\holiday.data
SET HDAY = MMMM
SET BUSDAYS = _MTWTF_
DATEADD(NEWDATE, 'BD', 1);
```

For 2011/11/02, DATEADD returns 2011/11/04 because November 3 is a holiday.

Example: Allocating the Holiday File to a Sequential File on z/OS Under PDS Deployment

The following sequential file, named USER1.HOLIDAY.DATA, defines November 3, 2011 and December 24, 2011 as holidays:

```
20111103
20111224
```

The following defines and sets the holiday file. Then DATEADD finds the next business day taking the holiday file into account:

```
DYNAM ALLOC DD HDAYMMMM DA USER1.HOLIDAY.DATA SHR REU
SET HDAY = MMMM
DATEADD(NEWDATE, 'BD', 1);
```

For 2011/11/02, DATEADD returns 2011/11/04 because November 3 is a holiday.

Example: Allocating the Holiday File to a PDS Member on z/OS Under PDS Deployment

The following holiday file, member HDAYMMMM in a PDS named USER1.HOLIDAY.DATA, defines November 3, 2011 and December 24, 2011 as holidays:

```
20111103
20111224
```

The following defines and sets the holiday file. Then DATEADD finds the next business day taking the holiday file into account:

```
DYNAM ALLOC DD HDAYMMMM DA USER1.HOLIDAY.DATA(HDAYMMMM) SHR REU
SET HDAY = MMMM
SET BUSDAYS = _MTWTF_
DATEADD(NEWDATE, 'BD', 1);
```

For 2011/11/02, DATEADD returns 2011/11/04 because November 3 is a holiday.

Enabling Leading Zeros For Date and Time Functions in Dialogue Manager

If you use a date and time function in Dialogue Manager that returns a numeric integer format, Dialogue Manager truncates any leading zeros. For example, if a function returns the value 000101 (indicating January 1, 2000), Dialogue Manager truncates the leading zeros, producing 101, an incorrect date. To avoid this problem, use the LEADZERO parameter.

LEADZERO only supports an expression that makes a direct call to a function. An expression that has nesting or another mathematical function always truncates leading zeros. For example,

```
-SET &OUT = AYM(&IN, 1, 'I4')/100;
```

truncates leading zeros regardless of the LEADZERO parameter setting.

Syntax: How to Set the Display of Leading Zeros

```
SET LEADZERO = {ON|OFF}
```

where:

`ON`

Displays leading zeros if present.

`OFF`

Truncates leading zeros. OFF is the default value.

Example: Displaying Leading Zeros

The AYM function adds one month to the input date of December 1999:

```
-SET &IN = '9912';
-RUN
-SET &OUT = AYM(&IN, 1, 'I4');
-TYPE &OUT
```

Using the default LEADZERO setting, this yields:

```
1
```

This represents the date January 2000 incorrectly. Setting the LEADZERO parameter in the request as follows:

```
SET LEADZERO = ON
-SET &IN = '9912';
-SET &OUT = AYM(&IN, 1, 'I4');
-TYPE &OUT
```

results in the following:

```
0001
```

This correctly indicates January 2000.

DATEADD: Adding or Subtracting a Date Unit to or From a Date

The DATEADD function adds a unit to or subtracts a unit from a full component date format. A unit is one of the following:

- ❏ **Year.**
- ❏ **Month.** If the calculation using the month unit creates an invalid date, DATEADD corrects it to the last day of the month. For example, adding one month to October 31 yields November 30, not November 31, since November has 30 days.
- ❏ **Day.**
- ❏ **Weekday.** When using the weekday unit, DATEADD does not count Saturday or Sunday. For example, if you add one day to Friday, first DATEADD moves to the next weekday, Monday, then it adds a day. The result is Tuesday.
- ❏ **Business day.** When using the business day unit, DATEADD uses the BUSDAYS parameter setting and holiday file to determine which days are working days and disregards the rest. If Monday is not a working day, then one business day past Sunday is Tuesday.

Note that when the DATEADD function calculates the next or previous business day or work day, it always starts from a business day or work day. So if the actual day is Saturday or Sunday, and the request wants to calculate the next business day, the function will use Monday as the starting day, not Saturday or Sunday, and will return Tuesday as the next business day. Similarly, when calculating the previous business day, it will use the starting day Friday, and will return Thursday as the previous business day. You can use the DATEMOV function to move the date to the correct type of day before using DATEADD.

DATEADD requires a date to be in date format. Since Dialogue Manager interprets a date as alphanumeric or numeric, and DATEADD requires a standard date stored as an offset from the base date, do not use DATEADD with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

Syntax: How to Add or Subtract a Date Unit to or From a Date

```
DATEADD(date, 'component', increment)
```

where:

date

Date

Is a full component date.

component

Alphanumeric

Is one of the following enclosed in single quotation marks:

`Y` indicates a year component.

`M` indicates a month component.

`D` indicates a day component.

`WD` indicates a weekday component.

`BD` indicates a business day component.

increment

Integer

Is the number of date units added to or subtracted from *date*. If this number is not a whole unit, it is rounded down to the next largest integer.

Note: DATEADD does not use an *output* argument. It uses the format of the *date* argument for the result. As long as the result is a full component date, it can be assigned only to a full component date field or to integer field.

Example: Adding or Subtracting a Date Unit to or From a Date

This example finds a delivery date that is 12 business days after today:

```
DELIV_DATE/YYMD = DATEADD('&DATEMDYY', 'BD', 12);
```

It returns 20040408, which will be Thursday if today is March 23 2004, Tuesday.

To make sure it is Thursday, assign it as

```
DELIV_DAY/W = DATEADD('&DATEMDYY', 'BD', 12);
```

which returns 4, representing Thursday. Note the use of the system variable &YYMD and the natural date representation of the today's date.

Tip: There is an alternative way to add to or subtract from the date. As long as any standard date is internally presented as a whole number of the least significant component units (that is, a number of days for full component dates, a number of months for YYM or MY format dates, and so on), you can add/subtract the desired number of these units directly, without DATEADD. Note that you must assign the date result to the same format date field, or the same field. For example, assuming YYM_DATE is a date field of format YYM, you can add 13 months to it and assign the result to the field NEW_YYM_DT, in the following statement:

```
NEW_YYM_DT/YYM = YYM_DATE + 13;
```

Otherwise, a non-full component date must be converted to a full component date before using DATEADD.

DATECVT: Converting the Format of a Date

The DATECVT function converts the field value of any standard date format or legacy date format into a date format (offset from the base date), in the desired standard date format or legacy date format. If you supply an invalid format, DATECVT returns a zero or a blank.

Syntax: How to Convert a Date Format

```
DATECVT(date, 'in_format', output)
```

where:

date

Date

Is the date to be converted. If you supply an invalid date, DATECVT returns zero. When the conversion is performed, a legacy date obeys any DEFCENT and YRTHRESH parameter settings supplied for that field.

in_format

Alphanumeric

Is the format of the date enclosed in single quotation marks. It is one of the following:

- ❏ A non-legacy date format (for example, YYMD, YQ, M, DMY, JUL).
- ❏ A legacy date format (for example, I6YMD or A8MDYY).
- ❏ A non-date format (such as I8 or A6). A non-date format in *in_format* functions as an offset from the base date of a YYMD field (12/31/1900).

output

Alphanumeric

Is the output format. It is one of the following:

- ❏ A non-legacy date format (for example, YYMD, YQ, M, DMY, JUL).
- ❏ A legacy date format (for example, I6YMD or A8MDYY).
- ❏ A non-date format (such as I8 or A6). This format type causes DATECVT to convert the date into a full component date and return it as a whole number in the format provided.

Example: Converting the Format of a Date

This example first converts a numeric date, NUMDATE, to a character date, and then assigns the result to a non-date alphanumeric field, CHARDATE.

```
CHARDATE/A13 = DATECVT (NUMDATE,'I8YYMD','A8YYMD');
```

Note: DATECVT does not use an output format; it uses the format of the argument output_format for the result.

DATEDIF: Finding the Difference Between Two Dates

The DATEDIF function returns the difference between two full component standard dates in units of a specified component. A component is one of the following:

- **Year.** Using the year unit with DATEDIF yields the inverse of DATEADD. If subtracting one year from date X creates date Y, then the count of years between X and Y is one. Subtracting one year from February 29 produces the date February 28.
- **Month.** Using the month component with DATEDIF yields the inverse of DATEADD. If subtracting one month from date X creates date Y, then the count of months between X and Y is one. If the to-date is the end-of-month, then the month difference may be rounded up (in absolute terms) to guarantee the inverse rule.

  If one or both of the input dates is the end of the month, DATEDIF takes this into account. This means that the difference between January 31 and April 30 is three months, not two months.
- **Day.**
- **Weekday.** With the weekday unit, DATEDIF does not count Saturday or Sunday when calculating days. This means that the difference between Friday and Monday is one day.
- **Business day.** With the business day unit, DATEDIF uses the BUSDAYS parameter setting and holiday file to determine which days are working days and disregards the rest. This means that if Monday is not a working day, the difference between Friday and Tuesday is one day. See *Rules for Creating a Holiday File* on page 210 for more information.

DATEDIF returns a whole number. If the difference between two dates is not a whole number, DATEDIF truncates the value to the next largest integer. For example, the number of years between March 2, 2001, and March 1, 2002, is zero. If the end date is before the start date, DATEDIF returns a negative number.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and DATEDIF requires a standard date stored as an offset from the base date, do not use DATEDIF with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

Syntax: How to Find the Difference Between Two Dates

```
DATEDIF('from_date', 'to_date', 'component')
```

where:

from_date

Date

Is the start date from which to calculate the difference. Is a full component date.

to_date

Date

Is the end date from which to calculate the difference.

component

Alphanumeric

Is one of the following enclosed in single quotation marks:

`Y` indicates a year unit.

`M` indicates a month unit.

`D` indicates a day unit.

`WD` indicates a weekday unit.

`BD` indicates a business day unit.

Note: DATEDIF does not use an *output* argument because for the result it uses the format 'I8'.

Example: Finding the Difference Between Two Dates

The example finds the number of complete months between today, March 23, 2004, and one specific day in the past

```
DATEDIF('September 11 2001', '20040323', 'M')
```

and returns 30, which can be assigned to a numeric field.

Tip: There is an alternative way to find the difference between dates. As long as any standard date is presented internally as a whole number of the least significant component units (that is, a number of days for full component dates, a number of months for YYM or MY format dates, etc.), you can find the difference in these component units (not any units) directly, without DATEDIF. For example, assume OLD_YYM_DT is a date field in format MYY and NEW_YYM_DT is another date in format YYM. Note that the least significant component for both formats is month, M. The difference in months, then, can be found by subtracting the field OLD_YYM_DT from NEW_YYM_DT in the following statement:

```
MYDIFF/I8 = NEW_YYM_DT/YYM - OLD_YYM_DT;
```

Otherwise, non-full component standard dates or legacy dates should be converted to full component standard dates before using DATEDIF.

DATEMOV: Moving a Date to a Significant Point

The DATEMOV function moves a date to a significant point on the calendar.

Note: Using the beginning of week point (BOW) will always return Monday, and using the end of week point (EOW) will always return Friday. Also, if the date used with the DATEMOV function falls on Saturday or Sunday, the actual date used by the function will be the moved forward to the next Monday. If you do not want to do the calculation by moving the date from Saturday or Sunday to Monday, or if you want the BOW to be Sunday and the EOW to be Saturday, you can use the DTRUNC function.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and DATEMOV requires a standard date stored as an offset from the base date, do not use DATEMOV with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date. For example, the following converts the integer legacy date 20050131 to a smart date, adds one month, and converts the result to an alphanumeric legacy date:

```
-SET &STRT=DATECVT(20050131,'I8YYMD', 'YYMD');
-SET &NMT=DATEADD(&STRT,'M',1);
-SET &NMTA=DATECVT(&NMT,'YYMD','A8MTDYY');
-TYPE A MONTH FROM 20050131 IS &NMTA
```

The output shows that the DATEADD function added the actual number of days in the month of February to get to the end of the month from the end of January:

```
A MONTH FROM 20050131 IS 02282005
```

DATEMOV works only with full component dates.

Syntax: How to Move a Date to a Significant Point

```
DATEMOV(date, 'move-point')
```

where:

date

Date

Is the date to be moved. It must be a full component format date (for example, MDYY or YYJUL).

move-point

Alphanumeric

Is the significant point the date is moved to enclosed in single quotation marks ('). An invalid point results in a return code of zero. Valid values are:

- **EOM,** which is the end of month.
- **BOM,** which is the beginning of month.
- **EOQ,** which is the end of quarter.
- **BOQ,** which is the beginning of quarter.
- **EOY,** which is the end of year.
- **BOY,** which is the beginning of year.
- **EOW,** which is the end of week.
- **BOW,** which is the beginning of week.
- **NWD,** which is the next weekday.
- **NBD,** which is the next business day.
- **PWD,** which is the prior weekday.
- **PBD,** which is the prior business day.
- **WD-,** which is a weekday or earlier.
- **BD-,** which is a business day or earlier.
- **WD+,** which is a weekday or later.
- **BD+,** which is a business day or later.

A business day calculation is affected by the BUSDAYS and HDAY parameter settings.

Note that when the DATEADD function calculates the next or previous business day or work day, it always starts from a business day or work day. So if the actual day is Saturday or Sunday, and the request wants to calculate the next business day, the function will use Monday as the starting day, not Saturday or Sunday, and will return Tuesday as the next business day. Similarly, when calculating the previous business day, it will use the starting day Friday, and will return Thursday as the previous business day.

To avoid skipping a business day or work day, use DATEMOV. To return the next business or work day, use BD- or WD- to first move to the previous business or work day (if it is already a business day or work day, it will not be moved). Then use DATEADD to move to the next business or work day. If you want to return the previous business or work day, first use BD+ or WD+ to move to the next business or work day (if it is already the correct type of day, it will not be moved). Then use DATEADD to return the previous business or work day.

Note: DATEMOV does not use an *output* argument. It uses the format of the *date* argument for the result. As long as the result is a full component date, it can be assigned only to a full component date field or to an integer field.

Example: Moving a Date to a Significant Point

This example finds the end day of the current date week

```
DATEDIF('&YYMD', 'EOW')
```

and returns 20040326 if today is 2004, March 23rd. Note the use of the system variable &YYMD and natural date representation in the first argument.

Example: Returning the Next Business Day

This example shows why you may need to use DATEMOV to get the correct result.

The following request against the GGSALES data source uses the BD (Business Day) move point against the DATE field. First DATE is converted to a smart date, then DATEADD is called with the BD move-point:

```
DEFINE FILE GGSALES
DT1/WMDYY=DATE;
DT2/WMDYY = DATEADD(DT1 ,'BD',1);
DAY/Dt = DT1;
 END

TABLE FILE GGSALES
SUM  DT1
DT2
BY DT1 NOPRINT
WHERE RECORDLIMIT EQ 10
 END
```

When the date is on a Saturday or Sunday on the output, the next business day is returned as a Tuesday. This is because before doing the calculation, the original date was moved to a business day:

```
  DT1               DT2
  ---               ---
  SUN, 09/01/1996  TUE, 09/03/1996
  FRI, 11/01/1996  MON, 11/04/1996
  SUN, 12/01/1996  TUE, 12/03/1996
  SAT, 03/01/1997  TUE, 03/04/1997
  TUE, 04/01/1997  WED, 04/02/1997
  THU, 05/01/1997  FRI, 05/02/1997
  SUN, 06/01/1997  TUE, 06/03/1997
  MON, 09/01/1997  TUE, 09/02/1997
  WED, 10/01/1997  THU, 10/02/1997
```

In the following version of the request, DATEMOV is called to make sure the starting day is a business day. The move point specified in the first call is BD- which only moves the date to the prior business day if it is not already a business day. The call to DATEADD then uses the BD move point to return the next business day:

```
DEFINE FILE GGSALES
DT1/WMDYY=DATE;
DT1A/WMDYY=DATEMOV(DT1, 'BD-');
DT2/WMDYY = DATEADD(DT1A,'BD',1);
DAY/Dt = DT1;
 END

TABLE FILE GGSALES
SUM  DT1 DT1A DT2
BY DT1 NOPRINT
WHERE RECORDLIMIT EQ 10
 END
```

On the output, the next business day after a Saturday or Sunday is now returned as Monday:

```
DT1               DT1A              DT2
  ---               ----              ---
  SUN, 09/01/1996  FRI, 08/30/1996  MON, 09/02/1996
  FRI, 11/01/1996  FRI, 11/01/1996  MON, 11/04/1996
  SUN, 12/01/1996  FRI, 11/29/1996  MON, 12/02/1996
  SAT, 03/01/1997  FRI, 02/28/1997  MON, 03/03/1997
  TUE, 04/01/1997  TUE, 04/01/1997  WED, 04/02/1997
  THU, 05/01/1997  THU, 05/01/1997  FRI, 05/02/1997
  SUN, 06/01/1997  FRI, 05/30/1997  MON, 06/02/1997
  MON, 09/01/1997  MON, 09/01/1997  TUE, 09/02/1997
  WED, 10/01/1997  WED, 10/01/1997  THU, 10/02/1997
```

Example: Using a DEFINE FUNCTION to Move a Date to the Beginning of the Week

The following DEFINE FUNCTION named BOWK takes a date and the name of the day you want to consider the beginning of the week and returns a date that corresponds to the beginning of the week:

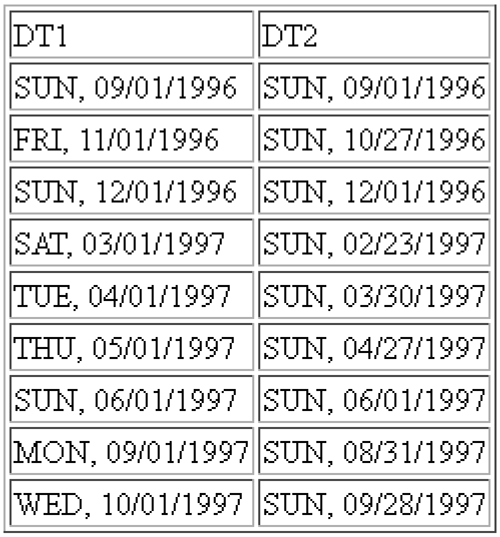
```
DEFINE FUNCTION BOWK(THEDATE/MDYY,WEEKSTART/A10)
DAYOFWEEK/W=THEDATE;
DAYNO/I1=IF DAYOFWEEK EQ 7 THEN 0 ELSE DAYOFWEEK;
FIRSTOFWK/I1=DECODE WEEKSTART('SUNDAY' 0 'MONDAY' 1 'TUESDAY' 2
'WEDNESDAY' 3 'THURSDAY' 4 'FRIDAY' 5 'SATURDAY' 6
'SUN' 0 'MON' 1 'TUE' 2 'WED' 3 'THU' 4 'FRI' 5 'SAT' 6);
BOWK/MDYY=IF DAYNO GE FIRSTOFWK THEN THEDATE-DAYNO+FIRSTOFWK
ELSE THEDATE-7-DAYNO+FIRSTOFWK;
END
```

The following request uses the BOWK function to use return a date (DT2) that corresponds to the beginning of the week for each value of the DT1 field:

```
DEFINE FILE GGSALES
DT1/WMDYY=DATE;
DT2/WMDYY = BOWK(DT1 ,'SUN');
 END

TABLE FILE GGSALES
SUM  DT1
DT2
BY DT1 NOPRINT
WHERE RECORDLIMIT EQ 10
ON TABLE SET PAGE NOLEAD
END
```

The output is shown in the following image:



DT1	DT2
SUN, 09/01/1996	SUN, 09/01/1996
FRI, 11/01/1996	SUN, 10/27/1996
SUN, 12/01/1996	SUN, 12/01/1996
SAT, 03/01/1997	SUN, 02/23/1997
TUE, 04/01/1997	SUN, 03/30/1997
THU, 05/01/1997	SUN, 04/27/1997
SUN, 06/01/1997	SUN, 06/01/1997
MON, 09/01/1997	SUN, 08/31/1997
WED, 10/01/1997	SUN, 09/28/1997

DATETRAN: Formatting Dates in International Formats

The DATETRAN function formats dates in international formats.

Syntax: How to Format Dates in International Formats

```
DATETRAN (indate, '(intype)', '([formatops])', 'lang', outlen, output)
```

where:

indate

Is the input date (in date format) to be formatted. Note that the date format cannot be an alphanumeric or numeric format with date display options (legacy date format).

intype

Is one of the following character strings indicating the input date components and the order in which you want them to display, enclosed in parentheses and single quotation marks.

The following table shows the single component input types:

Single Component Input Type	Description
`'(W)'`	Day of week component only (original format must have only W component).
`'(M)'`	Month component only (original format must have only M component).

The following table shows the two-component input types:

Two-Component Input Type	Description
`'(YYM)'`	Four-digit year followed by month.
`'(YM)'`	Two-digit year followed by month.
`'(MYY)'`	Month component followed by four-digit year.
`'(MY)'`	Month component followed by two-digit year.

The following table shows the three-component input types:

Three-Component Input Type	Description
`'(YYMD)'`	Four-digit year followed by month followed by day.
`'(YMD)'`	Two-digit year followed by month followed by day.
`'(DMYY)'`	Day component followed by month followed by four-digit year.

Three-Component Input Type	Description
`'(DMY)'`	Day component followed by month followed by two-digit year.
`'(MDYY)'`	Month component followed by day followed by four-digit year.
`'(MDY)'`	Month component followed by day followed by two-digit year.
`'(MD)'`	Month component followed by day (derived from three-component date by ignoring year component).
`'(DM)'`	Day component followed by month (derived from three-component date by ignoring year component).

formatops

Is a string of zero or more formatting options enclosed in parentheses and single quotation marks. The parentheses and quotation marks are required even if you do not specify formatting options. Formatting options fall into the following categories:

- ❑ Options for suppressing initial zeros in month or day numbers.

  Note: Zero suppression replaces initial zeros with blanks spaces.

- ❑ Options for translating month or day components to full or abbreviated uppercase or default case (mixed-case or lowercase depending on the language) names.
- ❑ Date delimiter options and options for punctuating a date with commas.

Valid options for suppressing initial zeros in month or day numbers are listed in the following table. Note that the initial zero is replaced by a blank space:

Format Option	Description
`m`	Zero-suppresses months (displays numeric months before October as 1 through 9 rather than 01 through 09).

Format Option	Description
`d`	Displays days before the tenth of the month as 1 through 9 rather than 01 through 09.
`dp`	Displays days before the tenth of the month as 1 through 9 rather than 01 through 09 with a period after the number.
`do`	Displays days before the tenth of the month as 1 through 9. For English (langcode EN) only, displays an ordinal suffix (st, nd, rd, or th) after the number.

The following table shows valid month and day name translation options:

Format Option	Description
`T`	Displays month as an abbreviated name, with no punctuation, all uppercase.
`TR`	Displays month as a full name, all uppercase.
`Tp`	Displays month as an abbreviated name, followed by a period, all uppercase.
`t`	Displays month as an abbreviated name with no punctuation. The name is all lowercase or initial uppercase, depending on language code.
`tr`	Displays month as a full name. The name is all lowercase or initial uppercase, depending on language code.
`tp`	Displays month as an abbreviated name, followed by a period. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).

Format Option	Description
`W`	Includes an abbreviated day-of-the-week name at the start of the displayed date, all uppercase with no punctuation.
`WR`	Includes a full day-of-the-week name at the start of the displayed date, all uppercase.
`Wp`	Includes an abbreviated day-of-the-week name at the start of the displayed date, all uppercase, followed by a period.
`w`	Includes an abbreviated day-of-the-week name at the start of the displayed date with no punctuation. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).
`wr`	Includes a full day-of-the-week name at the start of the displayed date. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).
`wp`	Includes an abbreviated day-of-the-week name at the start of the displayed date followed by a period. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).
`X`	Includes an abbreviated day-of-the-week name at the end of the displayed date, all uppercase with no punctuation.
`XR`	Includes a full day-of-the-week name at the end of the displayed date, all uppercase.

Format Option	Description
`Xp`	Includes an abbreviated day-of-the-week name at the end of the displayed date, all uppercase, followed by a period.
`x`	Includes an abbreviated day-of-the-week name at the end of the displayed date with no punctuation. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).
`xr`	Includes a full day-of-the-week name at the end of the displayed date. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).
`xp`	Includes an abbreviated day-of-the-week name at the end of the displayed date followed by a period. The name displays in the default case of the specified language (for example, all lowercase for French and Spanish, initial uppercase for English and German).

The following table shows valid date delimiter options:

Format Option	Description
`B`	Uses a blank as the component delimiter. This is the default if the month or day of week is translated or if comma is used.
`.`	Uses a period (.) as the component delimiter.
`-`	Uses a minus sign (-) as the component delimiter. This is the default when the conditions for a blank default delimiter are not satisfied.

Format Option	Description
/	Uses a slash (/) as the component delimiter.
|	Omits component delimiters.
K	Uses appropriate Asian characters as component delimiters.
c	Places a comma (,) after the month name (following T, Tp, TR, t, tp, or tr). Places a comma and blank after the day name (following W, Wp, WR, w, wp, or wr). Places a comma and blank before the day name (following X, XR, x, or xr).
e	Displays the Spanish or Portuguese word de or DE between the day and month, and between the month and year. The case of the word de is determined by the case of the month name. If the month is displayed in uppercase, DE is displayed. Otherwise, de is displayed. Useful for formats DMY, DMYY, MY, and MYY.
D	Inserts a comma (,) after the day number and before the general delimiter character specified.
Y	Inserts a comma (,) after the year and before the general delimiter character specified.

lang

Is the two-character standard ISO code for the language into which the date should be translated, enclosed in single quotation marks ('). Valid language codes are:

- 'AR' Arabic
- 'CS' Czech
- 'DA' Danish
- 'DE' German

- 'EN' English
- 'ES' Spanish
- 'FI' Finnish
- 'FR' French
- 'EL' Greek
- 'IW' Hebrew
- 'IT' Italian
- 'JA' Japanese
- 'KO' Korean
- 'LT' Lithuanian
- 'NL' Dutch
- 'NO' Norwegian
- 'PO' Polish
- 'PT' Portuguese
- 'RU' Russian
- 'SV' Swedish
- 'TH' Thai
- 'TR' Turkish
- 'TW' Chinese (Traditional)
- 'ZH' Chinese (Simplified)

outlen

Numeric

Is the length of the output field in bytes. If the length is insufficient, an all blank result is returned. If the length is greater than required, the field is padded with blanks on the right.

output

Alphanumeric

Reference: Usage Notes for the DATETRAN Function

- The output field, though it must be type A, and not A*n*V, may in fact contain variable length information, since the lengths of month names and day names can vary, and also month and day numbers may be either one or two bytes long if a zero-suppression option is selected. Unused bytes are filled with blanks.
- All invalid and inconsistent inputs result in all blank output strings. Missing data also results in blank output.
- The base dates (1900-12-31 and 1900-12 or 1901-01) are treated as though the DATEDISPLAY setting were ON (that is, not automatically shown as blanks). To suppress the printing of base dates, which have an internal integer value of 0, test for 0 before calling DATETRAN. For example:

  ```
  RESULT/A40 = IF DATE EQ 0 THEN ' ' ELSE
                  DATETRAN (DATE, '(YYMD)', '(.t)', 'FR', 40, 'A40');
  ```

- Valid translated date components are contained in files named DTLNG*lng* where *lng* is a three-character code that specifies the language. These files must be accessible for each language into which you want to translate dates.
- The DATETRAN function is not supported in Dialogue Manager.

Example: Using the DATETRAN Function

The following request prints the day of the week in the default case of the specific language:

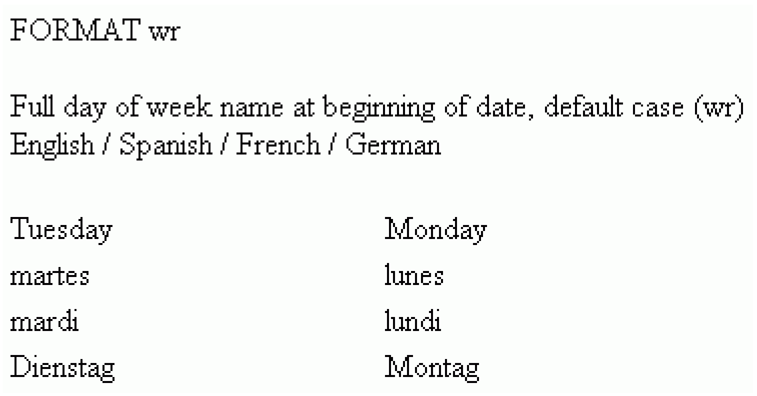
```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20051003;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1     ;
DATEYYMD2/YYMDW=TRANS2    ;

OUT1A/A8=DATETRAN(DATEW, '(W)', '(wr)', 'EN', 8 , 'A8') ;
OUT1B/A8=DATETRAN(DATEW2, '(W)', '(wr)', 'EN', 8 , 'A8') ;
OUT1C/A8=DATETRAN(DATEW, '(W)', '(wr)', 'ES', 8 , 'A8') ;
OUT1D/A8=DATETRAN(DATEW2, '(W)', '(wr)', 'ES', 8 , 'A8') ;
OUT1E/A8=DATETRAN(DATEW, '(W)', '(wr)', 'FR', 8 , 'A8') ;
OUT1F/A8=DATETRAN(DATEW2, '(W)', '(wr)', 'FR', 8 , 'A8') ;
OUT1G/A8=DATETRAN(DATEW, '(W)', '(wr)', 'DE', 8 , 'A8') ;
OUT1H/A8=DATETRAN(DATEW2, '(W)', '(wr)', 'DE', 8 , 'A8') ;
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT wr"
""
"Full day of week name at beginning of date, default case (wr)"
"English / Spanish / French / German"
""
SUM OUT1A AS '' OUT1B AS '' TRANSDATE NOPRINT
OVER OUT1C AS '' OUT1D AS ''
OVER OUT1E AS '' OUT1F AS ''
OVER OUT1G AS '' OUT1H AS ''
ON TABLE SET PAGE-NUM OFF
ON TABLE SET STYLE *
GRID=OFF, $
END
```

The output is:



```
FORMAT wr

Full day of week name at beginning of date, default case (wr)
English / Spanish / French / German

Tuesday                 Monday
martes                  lunes
mardi                   lundi
Dienstag                Montag
```

The following request prints a blank delimited date with an abbreviated month name in English. Initial zeros in the day number are suppressed, and a suffix is added to the end of the number:

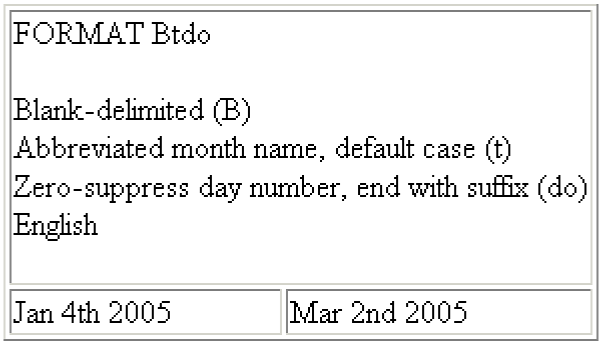
```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1     ;
DATEYYMD2/YYMDW=TRANS2    ;

OUT2A/A15=DATETRAN(DATEYYMD,  '(MDYY)', '(Btdo)', 'EN', 15, 'A15') ;
OUT2B/A15=DATETRAN(DATEYYMD2, '(MDYY)', '(Btdo)', 'EN', 15, 'A15') ;
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Btdo"
""
"Blank-delimited (B)"
"Abbreviated month name, default case (t)"
"Zero-suppress day number, end with suffix (do)"
"English"
""
SUM OUT2A AS '' OUT2B AS '' TRANSDATE NOPRINT
ON TABLE SET PAGE-NUM OFF
END
```

The output is:



```
FORMAT Btdo

Blank-delimited (B)
Abbreviated month name, default case (t)
Zero-suppress day number, end with suffix (do)
English

Jan 4th 2005    Mar 2nd 2005
```

The following request prints a blank delimited date, with an abbreviated month name in German. Initial zeros in the day number are suppressed, and a period is added to the end of the number:

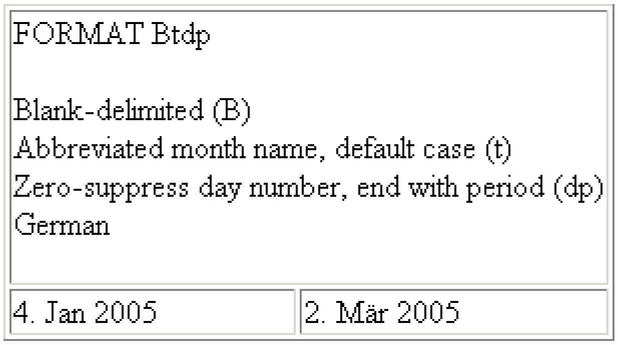
```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1    ;
DATEYYMD2/YYMDW=TRANS2   ;

OUT3A/A12=DATETRAN(DATEYYMD,  '(DMYY)', '(Btdp)', 'DE', 12, 'A12');
OUT3B/A12=DATETRAN(DATEYYMD2, '(DMYY)', '(Btdp)', 'DE', 12, 'A12');
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Btdp"
""
"Blank-delimited (B)"
"Abbreviated month name, default case (t)"
"Zero-suppress day number, end with period (dp)"
"German"
""
SUM OUT3A AS '' OUT3B AS '' TRANSDATE NOPRINT
ON TABLE SET PAGE-NUM OFF
END
```

The output is:



```
FORMAT Btdp

Blank-delimited (B)
Abbreviated month name, default case (t)
Zero-suppress day number, end with period (dp)
German

4. Jan 2005      2. Mär 2005
```

The following request prints a blank delimited date in French, with a full day name at the beginning and a full month name, in lowercase (the default for French):

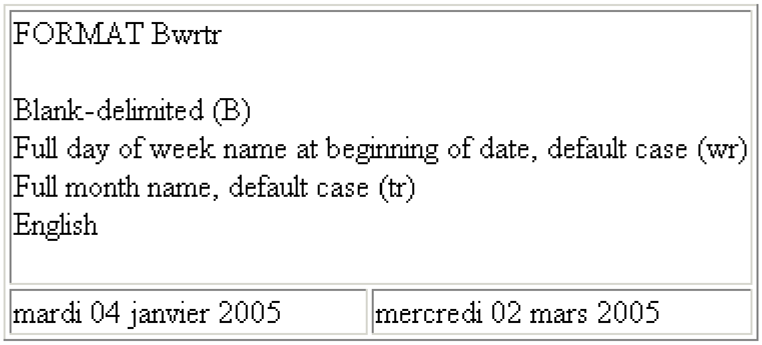
```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1     ;
DATEYYMD2/YYMDW=TRANS2    ;

OUT4A/A30 = DATETRAN(DATEYYMD,  '(DMYY)', '(Bwrtr)', 'FR', 30, 'A30');
OUT4B/A30 = DATETRAN(DATEYYMD2, '(DMYY)', '(Bwrtr)', 'FR', 30, 'A30');
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Bwrtr"
""
"Blank-delimited (B)"
"Full day of week name at beginning of date, default case (wr)"
"Full month name, default case (tr)"
"English"
""
SUM OUT4A AS '' OUT4B AS '' TRANSDATE NOPRINT
ON TABLE SET PAGE-NUM OFF
END
```

The output is:



FORMAT Bwrtr

Blank-delimited (B)
Full day of week name at beginning of date, default case (wr)
Full month name, default case (tr)
English

mardi 04 janvier 2005	mercredi 02 mars 2005

The following request prints a blank delimited date in Spanish with a full day name at the beginning in lowercase (the default for Spanish), followed by a comma, and with the word “de” between the day number and month and between the month and year:

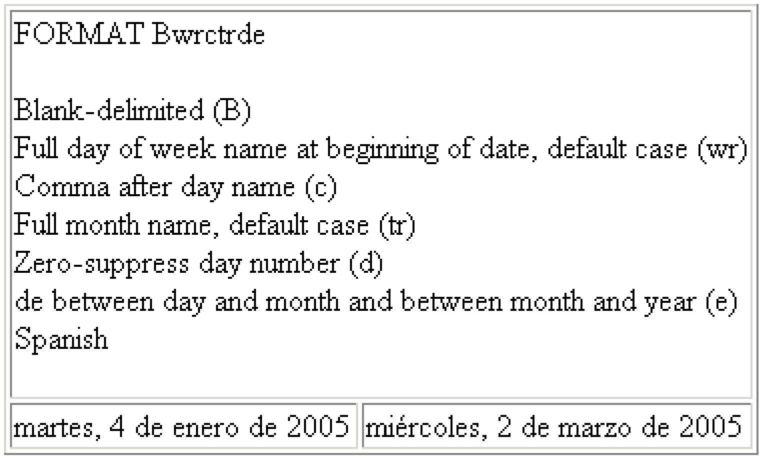
```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1     ;
DATEYYMD2/YYMDW=TRANS2    ;

OUT5A/A30=DATETRAN(DATEYYMD,  '(DMYY)', '(Bwrctrde)', 'ES', 30, 'A30');
OUT5B/A30=DATETRAN(DATEYYMD2, '(DMYY)', '(Bwrctrde)', 'ES', 30, 'A30');
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Bwrctrde"
""
"Blank-delimited (B)"
"Full day of week name at beginning of date, default case (wr)"
"Comma after day name (c)"
"Full month name, default case (tr)"
"Zero-suppress day number (d)"
"de between day and month and between month and year (e)"
"Spanish"
""
SUM OUT5A AS '' OUT5B AS '' TRANSDATE NOPRINT
ON TABLE SET PAGE-NUM OFF
END
```

The output is:



```
FORMAT Bwrctrde

Blank-delimited (B)
Full day of week name at beginning of date, default case (wr)
Comma after day name (c)
Full month name, default case (tr)
Zero-suppress day number (d)
de between day and month and between month and year (e)
Spanish

martes, 4 de enero de 2005    miércoles, 2 de marzo de 2005
```

The following request prints a date in Japanese characters with a full month name at the beginning, in the default case and with zero suppression:

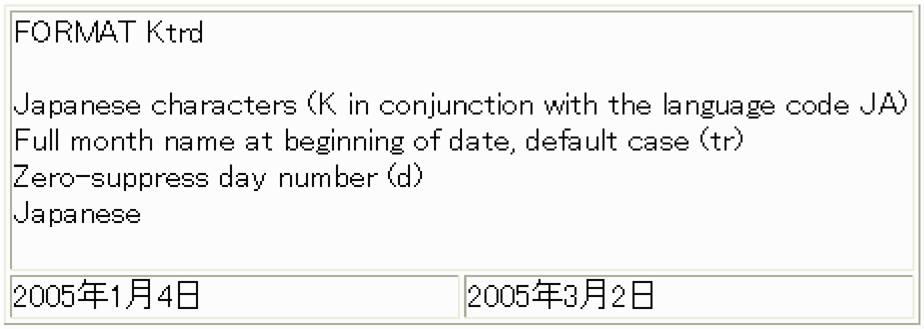
```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;

DATEW/W=TRANS1     ;
DATEW2/W=TRANS2    ;
DATEYYMD/YYMDW=TRANS1    ;
DATEYYMD2/YYMDW=TRANS2   ;

OUT6A/A30=DATETRAN(DATEYYMD , '(YYMD)', '(Ktrd)', 'JA', 30, 'A30');
OUT6B/A30=DATETRAN(DATEYYMD2, '(YYMD)', '(Ktrd)', 'JA', 30, 'A30');
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Ktrd"
""
"Japanese characters (K in conjunction with the language code JA)"
"Full month name at beginning of date, default case (tr)"
"Zero-suppress day number (d)"
"Japanese"
""
SUM OUT6A AS '' OUT6B AS '' TRANSDATE NOPRINT
ON TABLE SET PAGE-NUM OFF
END
```

The output is:



```
FORMAT Ktrd

Japanese characters (K in conjunction with the language code JA)
Full month name at beginning of date, default case (tr)
Zero-suppress day number (d)
Japanese

2005年1月4日                      2005年3月2日
```

The following request prints a blank delimited date in Greek with a full day name at the beginning in the default case, followed by a comma, and with a full month name in the default case:

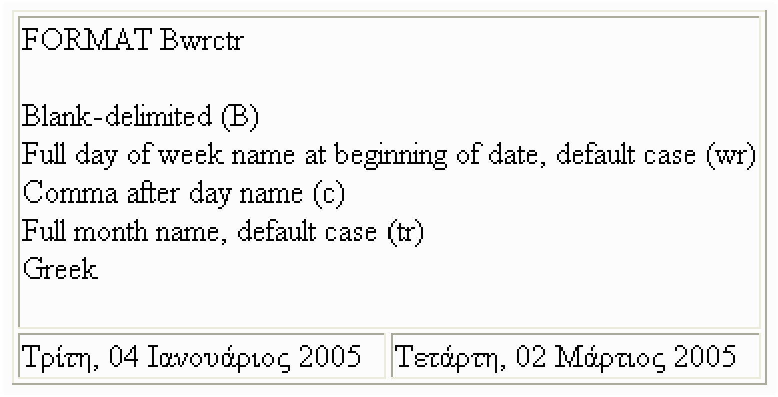
```
DEFINE FILE VIDEOTRK
TRANS1/YYMD=20050104;
TRANS2/YYMD=20050302;

DATEW/W=TRANS1      ;
DATEW2/W=TRANS2     ;
DATEYYMD/YYMDW=TRANS1     ;
DATEYYMD2/YYMDW=TRANS2    ;

OUT7A/A30=DATETRAN(DATEYYMD , '(DMYY)', '(Bwrctr)', 'GR', 30, 'A30');
OUT7B/A30=DATETRAN(DATEYYMD2, '(DMYY)', '(Bwrctr)', 'GR', 30, 'A30');
END

TABLE FILE VIDEOTRK
HEADING
"FORMAT Bwrctrde"
""
"Blank-delimited (B)"
"Full day of week name at beginning of date, default case (wr)"
"Comma after day name (c)"
"Full month name, default case (tr)"
"Greek"
""
SUM OUT7A AS '' OUT7B AS '' TRANSDATE NOPRINT
ON TABLE SET PAGE-NUM OFF
END
```

The output is:



FORMAT Bwrctr

Blank-delimited (B)
Full day of week name at beginning of date, default case (wr)
Comma after day name (c)
Full month name, default case (tr)
Greek

Τρίτη, 04 Ιανουάριος 2005	Τετάρτη, 02 Μάρτιος 2005

FIYR: Obtaining the Financial Year

The FIYR function returns the financial year, also known as the fiscal year, corresponding to a given calendar date based on the financial year starting date and the financial year numbering convention.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and FIYR requires a standard date stored as an offset from the base date, do not use FIYR with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

Syntax: How to Obtain the Financial Year

```
FIYR(inputdate, lowcomponent, startmonth, startday, yrnumbering, output)
```

where:

inputdate

Date

Is the date for which the financial year is returned. The date must be a standard date stored as an offset from the base date.

If the financial year does not begin on the first day of a month, the date must have Y(Y), M, and D components, or Y(Y) and JUL components (note that JUL is equivalent to YJUL). Otherwise, the date only needs Y(Y) and M components or Y(Y) and Q components.

lowcomponent

Alphanumeric

Is one of the following:

- `D` if the date contains a D or JUL component.
- `M` if the date contains an M component, but no D component.
- `Q` if the date contains a Q component.

`startmonth`

Numeric

1 through 12 are used to represent the starting month of the financial year, where 1 represents January and 12 represents December. If the low component is Q, the start month must be 1, 4, 7, or 10.

`startday`

Numeric

Is the starting day of the starting month, usually 1. If the low component is M or Q, 1 is required.

`yrnumbering`

Alphanumeric

Valid values are:

FYE to specify the *Financial Year Ending* convention. The financial year number is the calendar year of the ending date of the financial year. For example, when the financial year starts on October 1, 2008, the date 2008 November 1 is in FY 2009 Q1 because that date is in the financial year that ends on 2009 September 30.

FYS to specify the *Financial Year Starting* convention. The financial year number is the calendar year of the starting date of the financial year. For example, when the financial year starts on April 6, 2008, the date 2008 July 6 is in FY 2008 Q2 because that date is in the financial year that starts on 2008 April 6.

`output`

I, Y, or YY

The result will be in integer format, or Y or YY. This function returns a year value. In case of an error, zero is returned.

Note: February 29 cannot be used as a start day for a financial year.

Example: Obtaining the Financial Year

FIYR obtains the financial year for PERIOD, which has format YYM :

```
FIYR(PERIOD,'M', 4,1,'FYE','YY');
```

For PERIOD 2002/03, the result is 2002

For PERIOD 2002/04, the result is 2003.

FIQTR: Obtaining the Financial Quarter

The FIQTR function returns the financial quarter corresponding to a given calendar date based on the financial year starting date and the financial year numbering convention.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and FIQTR requires a standard date stored as an offset from the base date, do not use FIQTR with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

Syntax: How to Obtain the Financial Quarter

```
FIQTR(inputdate, lowcomponent, startmonth, startday, yrnumbering, output)
```

where:

inputdate

Date

Is the date for which the financial year is returned. The date must be a standard date stored as an offset from the base date.

If the financial year does not begin on the first day of a month, the date must have Y(Y), M, and D components, or Y(Y) and JUL components (note that JUL is equivalent to YJUL). Otherwise, the date only needs Y(Y) and M components or Y(Y) and Q components.

lowcomponent

Alphanumeric

Is one of the following:

- `D` if the date contains a D or JUL component.
- `M` if the date contains an M component, but no D component.
- `Q` if the date contains a Q component.

startmonth

Numeric

1 through 12 are used to represent the starting month of the financial year, where 1 represents January and 12 represents December. If the low component is Q, the start month must be 1, 4, 7, or 10.

startday

Numeric

Is the starting day of the starting month, usually 1. If the low component is M or Q, 1 is required.

yrnumbering

Alphanumeric

Valid values are:

FYE to specify the *Financial Year Ending* convention. The financial year number is the calendar year of the ending date of the financial year. For example, when the financial year starts on October 1, 2008, the date 2008 November 1 is in FY 2009 Q1 because that date is in the financial year that ends on 2009 September 30.

FYS to specify the *Financial Year Starting* convention. The financial year number is the calendar year of the starting date of the financial year. For example, when the financial year starts on April 6, 2008, the date 2008 July 6 is in FY 2008 Q2 because that date is in the financial year that starts on 2008 April 6.

output

I or Q

The result will be in integer format, or Q. This function will return a value of 1 through 4. In case of an error, zero is returned.

Note: February 29 cannot be used as a start day for a financial year.

Example: Obtaining the Financial Quarter

FIQTR obtains the financial quarter for START_DATE (format YYMD) and returns a column with format Q;

```
FIQTR(START_DATE,'D',10,1,'FYE','Q');
```

For 1997/10/01, the result is Q1.

For 1996/07/30, the result is Q4.

FIYYQ: Converting a Calendar Date to a Financial Date

The FIYYQ function returns a financial date containing both the financial year and quarter that corresponds to a given calendar date. The returned financial date is based on the financial year starting date and the financial year numbering convention.

Since Dialogue Manager interprets a date as alphanumeric or numeric, and FIYYQ requires a standard date stored as an offset from the base date, do not use FIYYQ with Dialogue Manager unless you first convert the variable used as the input date to an offset from the base date.

Syntax: How to Convert a Calendar Date to a Financial Date

```
FIYYQ(inputdate, lowcomponent, startmonth, startday, yrnumbering, output)
```

where:

`inputdate`

Date

Is the date for which the financial year is returned. The date must be a standard date stored as an offset from the base date.

If the financial year does not begin on the first day of a month, the date must have Y(Y), M, and D components, or Y(Y) and JUL components (note that JUL is equivalent to YJUL). Otherwise, the date only needs Y(Y) and M components or Y(Y) and Q components.

`lowcomponent`

Alphanumeric

Is one of the following:

- `D` if the date contains a D or JUL component.
- `M` if the date contains an M component, but no D component.
- `Q` if the date contains a Q component.

`startmonth`

Numeric

1 through 12 are used to represent the starting month of the financial year, where 1 represents January and 12 represents December. If the low component is Q, the start month must be 1, 4, 7, or 10.

`startday`

Numeric

Is the starting day of the starting month, usually 1. If the low component is M or Q, 1 is required.

`yrnumbering`

Alphanumeric

Valid values are:

FYE to specify the *Financial Year Ending* convention. The financial year number is the calendar year of the ending date of the financial year. For example, when the financial year starts on October 1, 2008, the date 2008 November 1 is in FY 2009 Q1 because that date is in the financial year that ends on 2009 September 30.

FYS to specify the *Financial Year Starting* convention. The financial year number is the calendar year of the starting date of the financial year. For example, when the financial year starts on April 6, 2008, the date 2008 July 6 is in FY 2008 Q2 because that date is in the financial year that starts on 2008 April 6.

`output`

Y[Y]Q or QY[Y]

In case of an error, zero is returned.

Note: February 29 cannot be used as a start day for a financial year.

Example: Converting a Calendar Date to a Financial Date

FIYYQ returns the financial date in format YQ that corresponds to START_DATE (format YYMD);

```
FIYYQ(START_DATE,'D',10,1,'FYE','YQ');
```

For 1997/10/01, the result is 98 Q1.

For 1996/07/30, the result is 96 Q4.

TODAY: Returning the Current Date

The TODAY function retrieves the current date from the operating system in the format MM/DD/YY or MM/DD/YYYY. It always returns a date that is current. Therefore, if you are running an application late at night, use TODAY. You can remove the default embedded slashes with the EDIT function.

You can also retrieve the date in the same format (separated by slashes) using the Dialogue Manager system variable &DATE. You can retrieve the date without the slashes using the system variables &YMD, &MDY, and &DMY. The system variable &DATE*fmt* retrieves the date in a specified format.

Syntax: How to Retrieve the Current Date

```
TODAY(output)
```

where:

output

Alphanumeric, at least A8

The following apply:

- If the format is A8 or A9, TODAY returns the 2-digit year.
- If the format is A10 or greater, TODAY returns the 4-digit year.

Example: Retrieving the Current Date

TODAY retrieves the current date and stores it in a column with the format A10.

```
TODAY('A10')
```

Using Legacy Date Functions

The legacy date functions were created for use with dates in integer, packed decimal, or alphanumeric format.

For detailed information on each legacy date function, see:

AYM: Adding or Subtracting Months

AYMD: Adding or Subtracting Days on page 249

CHGDAT: Changing How a Date String Displays on page 250

DA Functions: Converting a Legacy Date to an Integer on page 253

DMY, MDY, YMD: Calculating the Difference Between Two Dates on page 254

DOWK and DOWKL: Finding the Day of the Week on page 254

DT Functions: Converting an Integer to a Date on page 255

GREGDT: Converting From Julian to Gregorian Format on page 256

JULDAT: Converting From Gregorian to Julian Format on page 257

YM: Calculating Elapsed Months on page 258

Using Old Versions of Legacy Date Functions

The functions described in this section are legacy date functions. They were created for use with dates in integer or alphanumeric format. They are no longer recommended for date manipulation. Standard date and date-time functions are preferred.

All legacy date functions support dates for the year 2000 and later.

AYMD: Adding or Subtracting Days

The AYMD function adds days to or subtracts days from a date in year-month-day format. You can convert a date to this format using the CHGDAT or EDIT function.

Syntax: How to Add or Subtract Days to or From a Date

```
AYMD(indate, days, output)
```

where:

indate

I6, I6YMD, I8, I8YYMD

Is the legacy date in year-month-day format. If the date is not valid, the function returns the value 0 (zero).

days

Integer

Is the number of days you are adding to or subtracting from *indate*. To subtract days, use a negative number.

output

I6, I6YMD, I8, or I8YYMD

Is the same format as *indate*.

If the addition or subtraction of days crosses forward or backward into another century, the century digits of the output year are adjusted.

Example: Adding Days to a Date

AYMD adds 35 days to each value in the HIRE_DATE field, and stores the result in a column with the format I6YMD.

```
AYMD(HIRE_DATE, 35, 'I6YMD')
```

For 99/08/01, the result is 99/09/05.

For 99/01/04, the result is 99/02/08.

CHGDAT: Changing How a Date String Displays

The CHGDAT function rearranges the year, month, and day portions of an input character string representing a date. It may also convert the input string from long to short or short to long date representation. Long representation contains all three date components: year, month, and day; short representation omits one or two of the date components, such as year, month, or day. The input and output date strings are described by display options that specify both the order of date components (year, month, day) in the date string and whether two or four digits are used for the year (for example, 04 or 2004). CHGDAT reads an input date character string and creates an output date character string that represents the same date in a different way.

Note: CHGDAT requires a date character string as input, not a date itself. Whether the input is a standard or legacy date, convert it to a date character string (using the EDIT or DATECVT functions, for example) before applying CHGDAT.

The order of date components in the date character string is described by display options comprised of the following characters in your chosen order:

Character	Description
D	Day of the month (01 through 31).
M	Month of the year (01 through 12).
Y[Y]	Year. Y indicates a two-digit year (such as 94); YY indicates a four-digit year (such as 1994).

To spell out the month rather than use a number in the resulting string, append one of the following characters to the display options for the resulting string:

Character	Description
T	Displays the month as a three-letter abbreviation.
X	Displays the full name of the month.

Display options can consist of up to five display characters. Characters other than those display options are ignored.

For example: The display options 'DMYY' specify that the date string starts with a two digit day, then two digit month, then four digit year.

Note: Display options are *not* date formats.

Reference: Short to Long Conversion

If you are converting a date from short to long representation (for example, from year-month to year-month-day), the function supplies the portion of the date missing in the short representation, as shown in the following table:

Portion of Date Missing	Portion Supplied by Function
Day (for example, from YM to YMD)	Last day of the month.
Month (for example, from Y to YM)	Last month of the year (December).
Year (for example, from MD to YMD)	The year 99.
Converting year from two-digit to four-digit (for example, from YMD to YYMD)	The century will be determined by the 100-year window defined by DEFCENT and YRTHRESH.

Syntax: How to Change the Date Display String

```
CHGDAT('in_display_options','out_display_options',date_string,output)
```

where:

`'in_display_options'`

A1 to A5

Is a series of up to five display options that describe the layout of *date_string.* These options can be stored in an alphanumeric field or supplied as a literal enclosed in single quotation marks.

`'out_display_options'`

A1 to A5

Is a series of up to five display options that describe the layout of the converted date string. These options can be stored in an alphanumeric field or supplied as a literal enclosed in single quotation marks.

`date_string`

A2 to A8

Is the input date character string with date components in the order specified by *in_display_options*.

Note that if the original date is in numeric format, you must convert it to a date character string. If *date_string* does not correctly represent the date (the date is invalid), the function returns blank spaces.

`output`

A*xx*, where *xx* is a number of characters large enough to fit the date string specified by *out_display_options*. A17 is long enough to fit the longest date string.

Note: Since CHGDAT uses a date string (as opposed to a date) and returns a date string with up to 17 characters, use the EDIT or DATECVT functions or any other means to convert the date to or from a date character string.

Example: Converting the Date Display From YMD to MDYYX

ALPHA_HIRE is HIRE_DATE converted from numeric to alphanumeric format. CHGDAT converts each value in ALPHA_HIRE from displaying the components as YMD to MDYYX and stores the result in a column with the format A17. The option X in the output value displays the full name of the month.

```
CHGDAT('YMD', 'MDYYX', ALPHA_HIRE, 'A17')
```

DA Functions: Converting a Legacy Date to an Integer

The DA functions convert a legacy date to the number of days between it and a base date. By converting a date to the number of days, you can add and subtract dates and calculate the intervals between them, or you can add to or subtract numbers from the dates to get new dates.

You can convert the result back to a date using the DT functions discussed in *DT Functions: Converting an Integer to a Date* on page 255.

There are six DA functions; each one accepts a date in a different format.

Syntax: How to Convert a Date to an Integer

```
function(indate, output)
```

where:

`function`

Is one of the following:

`DADMY` converts a date in day-month-year format.

`DADYM` converts a date in day-year-month format.

`DAMDY` converts a date in month-day-year format.

`DAMYD` converts a date in month-year-day format.

`DAYDM` converts a date in year-day-month format.

`DAYMD` converts a date in year-month-day format.

`indate`

I6xxx or P6xxx, where xxx corresponds to the function DAxxx you are using.

Is the legacy date to be converted. If *indate* is a numeric literal, enter only the last two digits of the year; the function assumes the century component. If the date is invalid, the function returns a 0.

`output`

Integer

Example: Converting Dates and Calculating the Difference Between Them

DAYMD converts DAT_INC and HIRE_DATE to the number of days since December 31, 1899 and the smaller number is then subtracted from the larger number:

```
DAYMD(DAT_INC, 'I8') - DAYMD(HIRE_DATE, 'I8')
```

DMY, MDY, YMD: Calculating the Difference Between Two Dates

The DMY, MDY, and YMD functions calculate the difference between two legacy dates in integer, alphanumeric, or packed format.

Syntax: How to Calculate the Difference Between Two Dates

```
function(from_date, to_date)
```

where:

function

Is one of the following:

DMY calculates the difference between two dates in day-month-year format.

MDY calculates the difference between two dates in month-day-year format.

YMD calculates the difference between two dates in year-month-day format.

from_date

I, P, or A format with date display options.

Is the beginning legacy date.

to_date

I, P, or A format with date display options.I6xxx or I8xxx where xxx corresponds to the specified function (DMY, YMD, or MDY).

Is the end date.

Example: Calculating the Number of Days Between Two Dates

YMD calculates the number of days between the dates in HIRE_DATE and DAT_INC.

```
YMD(HIRE_DATE, DAT_INC)
```

DOWK and DOWKL: Finding the Day of the Week

The DOWK and DOWKL functions find the day of the week that corresponds to a date. DOWK returns the day as a three letter abbreviation; DOWKL displays the full name of the day.

Syntax: How to Find the Day of the Week

```
{DOWK|DOWKL}(indate, output)
```

where:

indate

I6YMD or I8YYMD

Is the legacy date in year-month-day format. If the date is not valid, the function returns spaces. If the date specifies a two digit year and DEFCENT and YRTHRESH values have not been set, the function assumes the 20th century.

output

DOWK: A4. DOWKL: A12

Example: Finding the Day of the Week

DOWK determines the day of the week that corresponds to the value in the HIRE_DATE field and stores the result in a column with the format A4.

```
DOWK(HIRE_DATE, 'A4')
```

For 80/06/02, the result is MON.

For 82/08/01, the result is SUN.

DT Functions: Converting an Integer to a Date

There are six DT functions; each one converts a number into a date of a different format.

Syntax: How to Convert an Integer to a Date

```
function(number, output)
```

where:

function

Is one of the following:

DTDMY converts a number to a day-month-year date.

DTDYM converts a number to a day-year-month date.

DTMDY converts a number to a month-day-year date.

DTMYD converts a number to a month-year-day date.

DTYDM converts a number to a year-day-month date.

DTYMD converts a number to a year-month-day date.

number

Integer

Is the number of days since the base date, possibly received from the functions DAxxx.

output

I8xxx, where xxx corresponds to the function DTxxx in the above list.

Example: Converting an Integer to a Date

DTMDY converts NEWF (which was converted to the number of days by DAYMD) to the corresponding date and stores the result in a column with the format I8MDYY.

```
DTMDY(NEWF, 'I8MDYY')
```

For 81/11/02, the result is 11/02/1981.

For 82/05/01, the result is 05/01/1982.

GREGDT: Converting From Julian to Gregorian Format

The GREGDT function converts a date in Julian format (year-day) to Gregorian format (year-month-day).

A date in Julian format is a five- or seven-digit number. The first two or four digits are the year; the last three digits are the number of the day, counting from January 1. For example, January 1, 1999 in Julian format is either 99001 or 1999001; June21, 2004 in Julian format is 2004173.

Syntax: How to Convert From Julian to Gregorian Format

```
GREGDT(indate, output)
```

where:

indate

I5 or I7

Is the Julian date. If the date is invalid, the function returns a 0 (zero).

output

I6, I8, I6YMD, or I8YYMD

Example: Converting From Julian to Gregorian Format

DTMDY converts NEWF (which was converted to the number of days by DAYMD) to the corresponding date and stores the result in a column with the format I8MDYY.

```
DTMDY(NEWF, 'I8MDYY')
```

For 81/11/02, the result is 11/02/1981.

For 82/05/01, the result is 05/01/1982.

JULDAT: Converting From Gregorian to Julian Format

The JULDAT function converts a date from Gregorian format (year-month-day) to Julian format (year-day). A date in Julian format is a five- or seven-digit number. The first two or four digits are the year; the last three digits are the number of the day, counting from January 1. For example, January 1, 1999 in Julian format is either 99001 or 1999001.

Syntax: How to Convert From Gregorian to Julian Format

```
JULDAT(indate, output)
```

where:

indate

I6, I8, I6YMD, I8YYMD

Is the legacy date to convert.

output

I5 or I7

Example: Converting From Gregorian to Julian Format

GREGDT converts JULIAN to YYMD (Gregorian) format. It determines the century using the default DEFCENT and YRTHRESH parameter settings. The result is stored in a column with the format I8.

```
GREGDT(JULIAN, 'I8')
```

For 82213, the result is 19820801.

For 82004, the result is 19820104.

YM: Calculating Elapsed Months

The YM function calculates the number of months between two dates. The dates must be in year-month format. You can convert a date to this format by using the CHGDAT or EDIT function.

Syntax: How to Calculate Elapsed Months

```
YM(fromdate, todate, output)
```

where:

fromdate

I4YM or I6YYM

Is the start date in year-month format (for example, I4YM). If the date is not valid, the function returns the value 0 (zero).

todate

I4YM or I6YYM

Is the end date in year-month format. If the date is not valid, the function returns the value 0 (zero).

output

Integer

Tip: If *fromdate* or *todate* is in integer year-month-day format (I6YMD or I8YYMD), simply divide by 100 to convert to year-month format and set the result to an integer. This drops the day portion of the date, which is now after the decimal point.

Example: Calculating Elapsed Months

YM calculates the difference between HIRE_MONTH and MONTH_INC and stores the results in a column with the format I3.

```
YM(HIRE_MONTH, MONTH_INC, 'I3')
```

Chapter 10

Date-Time Functions

Date-Time functions are for use with timestamps in date-time formats, also known as H formats. A timestamp value refers to internally stored data capable of holding both date and time components with an accuracy of up to a nanosecond.

In this chapter:

Using Date-Time Functions

The functions described in this section operate on fields in date-time format (sometimes called H format).

However, you can also provide a date as a character string using the macro DT, followed by a character string in parentheses, presenting date and time. Date components are separated by slashes '/'; time components by colons ':'.

Alternatively, the day can be given as a natural day, like 2004 March 31, in parentheses. Either the date or time component can be omitted. For example, the date-time format argument can be expressed as DT(2004/03/11 13:24:25.99) or DT(March 11 2004).

The following is another example that creates a timestamp representing the current date and time. The system variables &YYMD and &TOD are used to obtain the current date and time, respectively:

```
-SET &MYSTAMP = &YYMD | ' ' | EDIT(&TOD,'99:$99:$99') ;
```

Today's date (&YYMD) is concatenated with the time of day (&TOD). The EDIT function is used to change the dots (.) in the time of day variable to colons (:).

The following request uses the DT macro on the alphanumeric date and time variable &MYSTAMP:

```
TABLE FILE CAR
  PRINT CAR NOPRINT
  COMPUTE   DTCUR/HYYMDS = DT(&MYSTAMP);
  IF RECORDLIMIT IS 1;
END
```

Date-Time Parameters

The DATEFORMAT parameter specifies the order of the date components for certain types of date-time values. The WEEKFIRST parameter specifies the first day of the week. The DTSTRICT parameter determines the extent to which date-time values are checked for validity.

Specifying the Order of Date Components

The DATEFORMAT parameter specifies the order of the date components (month/day/year) when date-time values are entered in the formatted string and translated string formats . It makes the input format of a value independent of the format of the variable to which it is being assigned.

Syntax: How to Specify the Order of Date Components in a Date-Time Field

```
SET DATEFORMAT = option
```

where:

option

Can be one of the following: MDY, DMY, YMD, or MYD. MDY is the default value for the U.S. English format.

Specifying the First Day of the Week for Use in Date-Time Functions

The WEEKFIRST parameter specifies a day of the week as the start of the week. This is used in week computations by the HADD, HDIFF, HNAME, HPART, and HYYWD functions. It is also used by the DTADD, DTDIFF, DTRUNC, and DTPART functions. The default values are different for these functions, as described in *How to Set a Day as the Start of the Week* on page 263. The WEEKFIRST parameter does not change the day of the month that corresponds to each day of the week, but only specifies which day is considered the start of the week.

The HPART, DTPART, HYYWD, and HNAME subroutines can extract a week number from a date-time value. To determine a week number, they can use different definitions. For example, ISO 8601 standard week numbering defines the first week of the year as the first week in January with four or more days. Any preceding days in January belong to week 52 or 53 of the preceding year. The ISO standard also establishes Monday as the first day of the week.

You specify which type of week numbering to use by setting the WEEKFIRST parameter, as described in *How to Set a Day as the Start of the Week* on page 263.

Since the week number returned by HNAME, DTPART, and HPART functions can be in the current year or the year preceding or following, the week number by itself may not be useful. The function HYYWD returns both the year and the week for a given date-time value.

Syntax: How to Set a Day as the Start of the Week

```
SET WEEKFIRST = value
```

where:

value

Can be:

- **1 through 7**, representing Sunday through Saturday with non-standard week numbering.

Week numbering using these values establishes the first week in January with seven days as week number 1. Preceding days in January belong to the last week of the previous year. All weeks have seven days.

- **ISO1 through ISO7**, representing Sunday through Saturday with ISO standard week numbering.

  Note: ISO is a synonym for ISO2.

  Week numbering using these values establishes the first week in January with at least four days as week number 1. Preceding days in January belong to the last week of the previous year. All weeks have seven days.

- **STD1 through STD7**, in which the digit 1 (Sunday) through 7 (Saturday) indicates the starting day of the week.

  Note: STD without a digit is equivalent to STD1.

  Week numbering using these values is as follows. Week number 1 begins on January 1 and ends on the day preceding the first day of the week. For example, for STD1, the first week ends on the first Saturday of the year. The first and last week may have fewer than seven days.

- **SIMPLE**, which establishes January 1 as the start of week 1, January 8 is the start of week 2, and so on. The first day of the week is, thus, the same as the first day of the year. The last week (week 53) is either one or two days long.

- **0 (zero)**, is the value of the WEEKFIRST setting before the user issues an explicit WEEKFIRST setting. The date-time functions HPART, HNAME, HYYWD, HADD, and HDIFF use Saturday as the start of the week, when the WEEKFIRST setting is 0. The simplified functions DTADD, DTDIFF, DTRUNC, and DTPART, as well as printing of dates truncated to weeks, and recognition of date constant strings that contain week numbers, use Sunday as the default value, when the WEEKFIRST setting is 0. If the user explicitly sets WEEKFIRST to another value, that value is used by all of the functions.

Example: Setting Sunday as the Start of the Week

The following designates Sunday as the start of the week, using non-standard week numbering:

```
SET WEEKFIRST = 1
```

Syntax: How to View the Current Setting of WEEKFIRST

```
? SET WEEKFIRST
```

This returns the value that indicates the week numbering algorithm and the first day of the week. For example, the integer 1 represents Sunday with non-standard week numbering.

Controlling Processing of Date-Time Values

Strict processing checks date-time values when they are input by an end user, read from a transaction file, displayed, or returned by a subroutine to ensure that they represent a valid date and time. For example, a numeric month must be between 1 and 12, and the day must be within the number of days for the specified month.

Syntax: How to Enable Strict Processing of Date-Time Values

```
SET DTSTRICT = {ON|OFF}
```

where:

`ON`

Invokes strict processing. ON is the default value.

Strict processing checks date-time values when they are input by an end user, read from a transaction file, displayed, or returned by a subroutine to ensure that they represent a valid date and time. For example, a numeric month must be between 1 and 12, and the day must be within the number of days for the specified month.

If DTSTRICT is ON and the result would be an invalid date-time value, the function returns the value zero (0).

`OFF`

Does not invoke strict processing. Date-time components can have any value within the constraint of the number of decimal digits allowed in the field. For example, if the field is a two-digit month, the value can be 12 or 99, but not 115.

Supplying Arguments for Date-Time Functions

Date-time functions may operate on a component of a date-time value. This topic lists the valid component names and abbreviations for use with these functions.

Reference: Arguments for Use With Date and Time Functions

The following component names, valid abbreviations, and values are supported as arguments for the date-time functions that require them:

Component Name	Abbreviation	Valid Values
`year`	`yy`	0001-9999
`quarter`	`qq`	1-4
`month`	`mm`	1-12 or a month name, depending on the function.
`day-of-year`	`dy`	1-366
`day or day-of-month`	`dd`	1-31 (The two component names are equivalent.)
`week`	`wk`	1-53
`weekday`	`dw`	1-7 (Sunday-Saturday)
`hour`	`hh`	0-23
`minute`	`mi`	0-59
`second`	`ss`	0-59
`millisecond`	`ms`	0-999
`microsecond`	`mc`	0-999999
`nanosecond`	`ns`	0-999999999

Note:

❑ For an argument that specifies a length of eight, ten, or 12 characters, use eight to include milliseconds, ten to include microseconds, and 12 to include nanoseconds in the returned value.

- ❑ The last argument is always a USAGE format that indicates the data type returned by the function. The type may be A (alphanumeric), I (integer), D (floating-point double precision), H (date-time), or a date format (for example, YYMD).

HADD: Incrementing a Date-Time Value

The HADD function increments a date-time value by a given number of units.

Syntax: **How to Increment a Date-Time Value**

```
HADD(datetime, 'component', increment, length, output)
```

where:

datetime

Date-time

Is the date-time value to be incremented.

component

Alphanumeric

Is the name of the component to be incremented enclosed in single quotation marks. For a list of valid components, see *Arguments for Use With Date and Time Functions* on page 266.

Note: WEEKDAY is not a valid component for HADD.

increment

Integer

Is the number of units (positive or negative) by which to increment the component.

length

Integer

Is the number of characters returned. Valid values are:

- ❑ **8** indicates a date-time value that includes one to three decimal digits (milliseconds).
- ❑ **10** indicates a date-time value that includes four to six decimal digits (microseconds).
- ❑ **12** indicates a date-time value that includes seven to nine decimal digits (nanoseconds).

output

Date-time

Example: Incrementing a Date-Time Value

The following example increments thirty months to some specific date-time in the past

```
HADD(DT(2001/09/11 08:54:34), 'MONTH', 30, 8, 'HYYMDS')
```

and returns the timestamp 2004/03/11 08:54:34.00.

Example: Converting Unix (Epoch) Time to a Date-Time Value

Unix time (also known as Epoch time) defines an instant in time as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970, not counting leap seconds.

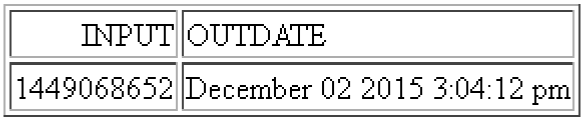
The following DEFINE FUNCTION takes a number representing epoch time and converts it to a date-time value by using the HADD function to add the number of seconds represented by the input value in epoch time to the epoch base date:

```
DEFINE FUNCTION UNIX2GMT(INPUT/I9)
   UNIX2GMT/HYYMDS = HADD(DT(1970 JAN 1),'SECONDS',INPUT,8,'HYYMDS');
END
```

The following request uses this DEFINE FUNCTION to convert the epoch time 1449068652 to a date-time value:

```
DEFINE FILE GGSALES
INPUT/I9=1449068652;
OUTDATE/HMTDYYSb = UNIX2GMT(INPUT);
END
TABLE FILE GGSALES
PRINT DATE NOPRINT INPUT OUTDATE
WHERE RECORDLIMIT EQ 1
ON TABLE SET PAGE NOLEAD
END
```

The output is shown in the following image:



INPUT	OUTDATE
1449068652	December 02 2015 3:04:12 pm

HCNVRT: Converting a Date-Time Value to Alphanumeric Format

The HCNVRT function converts a date-time value to alphanumeric format for use with operators such as EDIT, CONTAINS, and LIKE.

Syntax: How to Convert a Date-Time Value to Alphanumeric Format

```
HCNVRT(datetime, '(format)', length, output)
```

where:

datetime

Date-time

Is the date-time value to be converted.

format

Alphanumeric

Is the format of the date-time field enclosed in parentheses and single quotation marks. It must be a date-time format (data type H, up to H23).

length

Integer

Is the number of characters in the alphanumeric field that is returned. If *length* is smaller than the number of characters needed to display the alphanumeric field, the function returns a blank.

output

Alphanumeric

Example: Converting a Date-Time Value to Alphanumeric Format

Assume that you have a date-time field DTCUR in H format. To convert this timestamp to an alphanumeric string, use the following syntax:

```
HCNVRT(DTCUR, '(HMDYYS)', 20, 'A20')
```

The function returns the string '03/26/2004 14:25:58' that is assignable to an alphanumeric variable.

HDATE: Converting the Date Portion of a Date-Time Value to a Date Format

The HDATE function converts the date portion of a date-time value to the date format YYMD. You can then convert the result to other date formats.

Syntax: How to Convert the Date Portion of a Date-Time Value to a Date Format

```
HDATE(datetime, output)
```

where:

datetime

Date-time

Is the date-time value to be converted.

output

Date

Example: Converting the Date Portion of a Timestamp Value to a Date Format

This example converts the DTCUR field, which is the current date/time timestamp, into a date field using the format DMY:

```
MYDATE/DMY = HDATE(DTCUR, 'YYMD');
```

The function returns the date in format YYMD, then assigns it to MYDATE after conversion to its format MY as 03/04. Note that the output_format of HDATE is presented as a full component date format MDYY, as required.

HDIFF: Finding the Number of Units Between Two Date-Time Values

The HDIFF function calculates the number of date or time component units between two date-time values.

Reference: Usage Notes for HDIFF

HDIFF does its subtraction differently from DATEDIF, which subtracts date components stored in date fields. The DATEDIF calculation looks for full years or full months. Therefore, subtracting the following two dates and requesting the number of months or years, results in 0:

```
DATE1 12/25/2014, DATE2 1/5/2015
```

Performing the same calculation using HDIFF on date-time fields results in a value of 1 month or 1 year as, in this case, the month or year is first extracted from each date-time value, and then the subtraction occurs.

Syntax: How to Find the Number of Units Between Two Date-Time Values

```
HDIFF(end_dt, start_dt, 'component', output)
```

where:

end_dt

Date-time

Is the date-time value to subtract from.

start_dt

Date-time

Is the date-time value to subtract.

component

Alphanumeric

Is the name of the component to be used in the calculation, enclosed in single quotation marks. If the component is a week, the WEEKFIRST parameter setting is used in the calculation.

output

Floating-point double-precision

Example: Finding the Number of Units Between Two Date-Time Values

Assume that we have a date-time field DTCUR in H format, which is has a current date and time timestamp. To find the number of days from President's Day 2004 to today use the expression:

```
DIFDAY/I6 = HDIF(DTCUR, DT(2004/02/16), 'DAY', 'D6.0')
```

The function returns the number of days in double precision floating point format, then assigns it to DIFDAY as integer value. If today is March 31, 2004, the DIFDAY is assigned to 46.

If you wish to obtain results in seconds, use the expression

```
DIFSEC/I9 = HDIF(DTCUR, DT(2004 February 16), 'SECOND', 'D9.0')
```

which assigns 3801600 to DIFSEC. Note that the format 'D9.0' is used with HDIF. Using 'I9' for an output_format in HDIF is invalid.

HDTTM: Converting a Date Value to a Date-Time Value

The HDTTM function converts a date value to a date-time value. The time portion is set to midnight.

Syntax: How to Convert a Date Value to a Date-Time Value

```
HDTTM(date, length, output)
```

where:

date

Date

Is the date to be converted. It must be a full component format date. For example, it can be MDYY or YYJUL.

length

Integer

Is the length of the returned date-time value. Valid values are:

- **8** indicates a time value that includes milliseconds.
- **10** indicates a time value that includes microseconds.
- **12** indicates a time value that includes nanoseconds.

output

Date-time

Is the generated date-time value. The value must have a date-time format (data type H).

Example: Converting a Date to a Timestamp

This example converts the President's Day date into a timestamp:

```
TS/HYYMDS = HDTTM('February 16 2004', 8, TS)
```

the function returns 2004/02/16 00:00:00 and assigns this timestamp to field TS. Note the zero values of time components in the timestamp. Also note the use of natural date constants in single quotation marks for the date in the first function parameter.

HGETC: Storing the Current Local Date and Time in a Date-Time Field

The HGETC function returns the current local date and time in the desired date-time format. If millisecond or microsecond values are not available in your operating environment, the function retrieves the value zero for these components.

Syntax: How to Store the Current Local Date and Time in a Date-Time Field

```
HGETC(length, output)
```

where:

length

Integer

Is the length of the returned date-time value. Valid values are:

- **8** indicates a time value that includes milliseconds.
- **10** indicates a time value that includes microseconds.
- **12** indicates a time value that includes nanoseconds.

output

Date-time

Is the returned date-time value.

Example: Storing the Current Date and Time as a Timestamp

This example,

```
HGETC(8, 'HYYMDS')
```

creates a timestamp representing the current date and time.

HGETZ: Storing the Current Coordinated Universal Time in a Date-Time Field

HGETZ provides the current Coordinated Universal Time (UTC/GMT time, often called Zulu time). UTC is the primary civil time standard by which the world regulates clocks and time.

The value is returned in the desired date-time format. If millisecond or microsecond values are not available in your operating environment, the function retrieves the value zero for these components.

Syntax: How to Store the Current Universal Date and Time in a Date-Time Field

```
HGETZ(length, output)
```

where:

length

Integer

Is the length of the returned date-time value. Valid values are:

- ❑ **8** indicates a time value that includes milliseconds.
- ❑ **10** indicates a time value that includes microseconds.
- ❑ **12** indicates a time value that includes nanoseconds.

output

Date-time

Is the returned date-time value.

Example: Storing the Current Universal Date and Time as a Timestamp

This example,

```
HGETZ(8, 'HYYMDS')
```

creates a timestamp representing the current date and time.

Example: Calculating the Time Zone

The time zone can be calculated as a positive or negative hourly offset from GMT. Locations to the west of the prime meridian have a negative offset. The following request uses the HGETC function to retrieve the local time, and the HGETZ function to retrieve the GMT time. The HDIFF function calculates the number of boundaries between them in minutes. The zone is found by dividing the minutes by 60:

```
DEFINE FILE EMPLOYEE
LOCALTIME/HYYMDS = HGETC(8, LOCALTIME);
UTCTIME/HYYMDS = HGETZ(8, UTCTIME);
MINUTES/D4= HDIFF(LOCALTIME, UTCTIME, 'MINUTES', 'D4');
ZONE/P3 = MINUTES/60;
END
TABLE FILE EMPLOYEE
PRINT EMP_ID NOPRINT OVER
LOCALTIME  OVER
UTCTIME OVER
MINUTES OVER
ZONE
IF RECORDLIMIT IS 1
END
```

The output is:

```
  LOCALTIME  2015/05/12 12:47:04
  UTCTIME    2015/05/12 16:47:04
  MINUTES                   -240
  ZONE                        -4
```

HHMMSS: Retrieving the Current Time

The HHMMSS function retrieves the current time from the operating system as an eight character string, separating the hours, minutes, and seconds with periods.

Syntax: How to Retrieve the Current Time

```
HHMMSS(output)
```

where:

`output`

Alphanumeric, at least A8

Example: **Retrieving the Current Time**

This example,

```
HMMSS('A10')
```

creates a character string representing current time, like 12.09.47. Note that shorter output_format format will cause truncation of output.

HHMS: Converting a Date-Time Value to a Time Value

The HHMS function converts a date-time value to a time value.

Syntax: **How to Convert a Date-Time Value to a Time Value**

```
HHMS(datetime, length, output)
```

where:

datetime

Date-time

Is the date-time value to be converted.

length

Numeric

Is the length of the returned time value. Valid values are:

- ❑ **8** indicates a time value that includes milliseconds.
- ❑ **10** indicates a time value that includes microseconds.
- ❑ **12** indicates a time value that includes nanoseconds.

output

Time

Example: **Converting a Date-Time Value to a Time value**

HHMS converts the date-time field TRANSDATE to a time value with format HHIS:

```
HHMS(TRANSDATE, 8, 'HHIS')
```

For 2000/06/26 05:45, the output is 05:45:00

HINPUT: Converting an Alphanumeric String to a Date-Time Value

The HINPUT function converts an alphanumeric string to a date-time value.

Syntax: How to Convert an Alphanumeric String to a Date-Time Value

```
HINPUT(source_length, 'source_string', output_length, output)
```

where:

source_length

Integer

Is the number of characters in the source string to be converted.

source_string

Alphanumeric

Is the string to be converted.

output_length

Integer

Is the length of the returned date-time value. Valid values are:

- **8** indicates a time value that includes one to three decimal digits (milliseconds).
- **10** indicates a time value that includes four to six decimal digits (microseconds).
- **12** indicates a time value that includes seven to nine decimal digits (nanoseconds).

output

Date-time

Is the returned date-time value.

Example: Converting an Alphanumeric String to a Timestamp

This example,

```
DTM/HYYMDS = HINPUT(14, '20040229 13:34:00', 8, DTM);
```

converts the character string (20040229 13:34:00) into a timestamp, which is then assigned to the date-time field DTM. DTM is displayed as 2004/02/29 13:34:00.

HMIDNT: Setting the Time Portion of a Date-Time Value to Midnight

The HMIDNT function changes the time portion of a date-time value to midnight (all zeros by default). This allows you to compare a date field with a date-time field.

Syntax: How to Set the Time Portion of a Date-Time Value to Midnight

```
HMIDNT(datetime, length, output)
```

where:

datetime

Date-time

Is the date-time value whose time is to be set to midnight.

length

Integer

Is the length of the returned date-time value. Valid values are:

- **8** indicates a time value that includes milliseconds.
- **10** indicates a time value that includes microseconds.
- **12** indicates a time value that includes nanoseconds.

output

Date-time

Is the date-time return value whose time is set to midnight and whose date is copied from timestamp.

Example: Setting the Time Portion of a Timestamp to Midnight

This example converts the character string (20040229 13:34:00) to a timestamp, which is assigned to DTM:

```
DTM/HYYMDS = HINPUT(14, '20040229 13:34:00', 8, DTM);
```

This example resets the time portion of DTM to midnight and assigned the timestamp (02/29/2004 00:00:00) to DTMIDNT:

```
DTMIDNT/HMDYYS = HMIDNT(DTM, 8, DTMIDNT);
```

HNAME: Retrieving a Date-Time Component in Alphanumeric Format

The HNAME function extracts a specified component from a date-time value and returns it in alphanumeric format.

Syntax: **How to Retrieve a Date-Time Component in Alphanumeric Format**

```
HNAME(datetime, 'component', output)
```

where:

datetime

Date-time

Is the date-time value from which a component value is to be extracted.

component

Alphanumeric

Is the name of the component to be retrieved enclosed in single quotation marks. For a list of valid components, see *Arguments for Use With Date and Time Functions* on page 266.

output

Alphanumeric, at least A2

The function converts a month argument to an abbreviation of the month name and converts and all other components to strings of digits only. The year is always four digits, and the hour assumes the 24-hour system.

Example: **Retrieving a Timestamp Date or Time Component as an Alphanumeric Value**

Assuming that the current time obtained by the function HGETC in the first parameter is 13:22:11, this example returns the string '13' and assigns it to AHOUR:

```
AHOUR/A2 = HNAME(HGETC(8,'HYYMDS'),'HOUR', AHOUR);
```

Example: **Retrieving a Timestamp Date or Time Component as an Alphanumeric Value**

Assuming that the current time obtained by the function HGETC in the first parameter is 13:22:11, this example returns the string '13' and assigns it to AHOUR:

```
AHOUR/A2 = HNAME(HGETC(8,'HYYMDS'),'HOUR', AHOUR);
```

HPART: Retrieving a Date-Time Component as a Numeric Value

The HPART function extracts a specified component from a date-time value and returns it in numeric format.

Syntax: How to Retrieve a Date-Time Component in Numeric Format

```
HPART(datetime, 'component', output)
```

where:

`datetime`

Date-time

Is the date-time value from which the component is to be extracted.

`component`

Alphanumeric

Is the name of the component to be retrieved enclosed in single quotation marks. For a list of valid components, see *Arguments for Use With Date and Time Functions* on page 266.

`output`

Integer

Example: Retrieving a Timestamp Date or Time Component as Numeric Value

Assuming that the current time obtained by HGETC in the first parameter is 14:01:39, this example returns a whole number, 14, and assigns it to IHOUR:

```
IHOUR/I2 = HPART(HGETC(8,'HYYMDS'),'HOUR', IHOUR);
```

HSETPT: Inserting a Component Into a Date-Time Value

The HSETPT function inserts the numeric value of a specified component into a date-time value.

Syntax: How to Insert a Component Into a Date-Time Value

```
HSETPT(datetime, 'component', value, length, output)
```

where:

`datetime`

Date-time

Is the date-time value in which to insert the component.

component

Alphanumeric

Is the name of the component to be inserted enclosed in single quotation marks. See *Arguments for Use With Date and Time Functions* on page 266 for a list of valid components.

value

Integer

Is the numeric value to be inserted for the requested component.

length

Integer

Is the length of the returned date-time value. Valid values are:

- **8** indicates a time value that includes one to three decimal digits (milliseconds).
- **10** indicates a time value that includes four to six decimal digits (microseconds).
- **12** indicates a time value that includes seven to nine decimal digits (nanoseconds).

output

Date-time

Is the returned date-time value whose chosen component is updated. All other components are copied from the source date-time value.

Example: Inserting a Component Into a Date-Time Value

Assuming that the current date and time obtained by HGETC in the first parameter are 03/31/2004 and 13:34:36, this example,

```
UHOUR/HMDYYS = HSETPT(HGETC(8,'HYYMDS'),'HOUR', 7, 8, UHOUR);
```

returns 03/31/2004 07:34:36.

HTIME: Converting the Time Portion of a Date-Time Value to a Number

The HTIME function converts the time portion of a date-time value to the number of milliseconds if the length argument is eight, microseconds if the length argument is ten, or nanoseconds if the length argument is 12.

Syntax: How to Convert the Time Portion of a Date-Time Value to a Number

```
HTIME(length, datetime, output)
```

where:

length

Integer

Is the length of the input date-time value. Valid values are:

- ❑ **8** indicates a time value that includes one to three decimal digits (milliseconds).
- ❑ **10** indicates a time value that includes four to six decimal digits (microseconds).
- ❑ **12** indicates a time value that includes seven to nine decimal digits (nanoseconds).

datetime

Date-time

Is the date-time value from which to convert the time.

output

Floating-point double-precision

Example: Converting the Time Portion of a Date-Time Value to a Number

Assuming that the current date and time obtained by HGETC in the second parameter are 03/31/2004 and 13:48:14, this example returns and assigns to NMILLI, 49,694,395. (Note that this example uses milliseconds rather than microseconds.)

```
NMILLI/D12.0 = HTIME(8, HGETC(10,'HYYMDS'), NMICRO);
```

Assuming that the first parameter is equal to 10 and the timestamp format is HYYMDSS, this example returns and assigns to NMICRO, 50,686,123,024.

```
NMICRO/D12.0 = HTIME(10, HGETC(10,'HYYMDSS'), NMICRO);
```

HTMTOTS: Converting a Time to a Timestamp

The HTMTOTS function returns a timestamp using the current date to supply the date components of its value, and copies the time components from its input date-time value.

Syntax: How to Convert a Time to a Timestamp

```
HTMTOTS(time, length, output)
```

where:

time

Date-Time

Is the date-time value whose time will be used. The date portion will be ignored.

length

Integer

Is the length of the result. This can be one of the following:

- **8** for input time values including milliseconds.
- **10** for input time values including microseconds.
- **12** for input time values including nanoseconds.

output_format

Date-Time

Is the timestamp whose date is set to the current date, and whose time is copied from time.

Example: Converting a Time to a Timestamp

This example produces a timestamp, whose date and time are current, and stores the result in a column with the format in the field HMDYYS:

```
HMDYYS = HTMTOTS(DT(&MYTOD), 8, 'HMDYYS');
```

The result is 03/26/2004 13:48:14.

HYYWD: Returning the Year and Week Number From a Date-Time Value

The week number returned by HNAME and HPART can actually be in the year preceding or following the input date.

The HYYWD function returns both the year and the week number from a given date-time value.

The output is edited to conform to the ISO standard format for dates with week numbers, yyyy-Www-d.

Syntax: How to Return the Year and Week Number From a Date-Time Value

```
HYYWD(dtvalue, output)
```

where:

dtvalue

Date-time

Is the date-time value to be edited.

output

Alphanumeric

The output format must be at least 10 characters long. The output is in the following format:

```
yyyy-Www-d
```

where:

`yyyy`

Is the four-digit year.

`ww`

Is the two-digit week number (01 to 53).

`d`

Is the single-digit day of the week (1 to 7). The d value is relative to the current WEEKFIRST setting. If WEEKFIRST is 2 or ISO2 (Monday), then Monday is represented in the output as 1, Tuesday as 2.

Using the EDIT function, you can extract the individual subfields from this output.

Example: Returning the Year and Week Number From a Date-time Value

The following converts the TRANSDATE date-time value to the ISO standard format for dates with week numbers. WEEKFIRST is set to ISO2, which produces ISO standard week numbering:

```
 ISODATE/A10 = HYYWD(TRANSDATE, 'A10');
```

For date component 1999/01/30 04:16, the value is 1999-W04-6.

For date component 1999/12/15, the value is 1999-W50-3.

Chapter 11

Simplified Conversion Functions

Simplified conversion functions have streamlined parameter lists, similar to those used by SQL functions. In some cases, these simplified functions provide slightly different functionality than previous versions of similar functions.

The simplified functions do not have an output argument. Each function returns a value that has a specific data type.

When used in a request against a relational data source, these functions are optimized (passed to the RDBMS for processing).

In this chapter:

CHAR: Returning a Character Based on a Numeric Code

The CHAR function accepts a decimal integer and returns the character identified by that number converted to ASCII or EBCDIC, depending on the operating environment. The output is returned as variable length alphanumeric. If the number is above the range of valid characters, a null value is returned.

For a chart of printable characters and their decimal equivalents, see *Character Chart for ASCII and EBCDIC*.

Syntax: How to Return a Character Based on a Numeric Code

```
CHAR(number_code)
```

where:

number_code

Integer

Is a field, number, or numeric expression whose whole absolute value will be used as a number code to retrieve an output character.

For example, a TAB character is returned by CHAR(9) in ASCII environments, or by CHAR(5) in EBCDIC environments.

Example: Using the CHAR Function to Insert Control Characters Into a String

CHAR returns a carriage control character in an ASCII environment.

```
CHAR(13)
```

COMPACTFORMAT: Displaying Numbers in an Abbreviated Format

COMPACTFORMAT displays numbers in a compact format where:

- K is an abbreviation for thousands.
- M is an abbreviation for millions.
- B is an abbreviation for billions.
- T is an abbreviation for trillions.

COMPACTFORMAT computes which abbreviation to use, based on the order of magnitude of the largest value in the column. The returned value is an alphanumeric string. Attempting to output this value to a numeric format will result in a format error, and the value zero (0) will be displayed.

Syntax: How to Display Numbers in an Abbreviated Format

```
COMPACTFORMAT(input)
```

where:

input

Is the name of a numeric field.

Example: **Displaying Numbers in an Abbreviated Format**

COMPACTFORMAT abbreviates the display of COGS_US.

```
COMPACTFORMAT(COGS_US)
```

For $2,950,358.00, the result is $3M.

CTRLCHAR: Returning a Non-Printable Control Character

The CTRLCHAR function returns a nonprintable control character specific to the running operating environment, based on a supported list of keywords. The output is returned as variable length alphanumeric.

Syntax: **How to Return a Non-Printable Control Character**

```
CTRLCHAR(ctrl_char)
```

where:

ctrl_char

Is one of the following keywords.

- **NUL** returns a null character.
- **SOH** returns a start of heading character.
- **STX** returns a start of text character.
- **ETX** returns an end of text character.
- **EOT** returns an end of transmission character.
- **ENQ** returns an enquiry character.
- **ACK** returns an acknowledge character.
- **BEL** returns a bell or beep character.
- **BS** returns a backspace character.
- **TAB** or **HT** returns a horizontal tab character.
- **LF** returns a line feed character.
- **VT** returns a vertical tab character.
- **FF** returns a form feed (top of page) character.

- **CR** returns a carriage control character.
- **SO** returns a shift out character.
- **SI** returns a shift in character.
- **DLE** returns a data link escape character.
- **DC1** or **XON** returns a device control 1 character.
- **DC2** returns a device control 2 character.
- **DC3** or **XOFF** returns a device control 3 character.
- **DC4** returns a device control 4 character.
- **NAK** returns a negative acknowledge character.
- **SYN** returns a synchronous idle character.
- **ETB** returns an end of transmission block character.
- **CAN** returns a cancel character.
- **EM** returns an end of medium character.
- **SUB** returns a substitute character.
- **ESC** returns an escape, prefix, or altmode character.
- **FS** returns a file separator character.
- **GS** returns a group separator character.
- **RS** returns a record separator character.
- **US** returns a unit separator character.
- **DEL** returns a delete, rubout, or interrupt character.

Example: Using the CTRLCHAR Function to Insert Control Characters Into a String

CTRLCHAR returns a carriage control character in an ASCII environment.

```
CTRLCHAR(CR)
```

DT_FORMAT: Converting a Date or Date-Time Value to an Alphanumeric String

DT_FORMAT converts a date or date-time value to an alphanumeric string in a specified date or date-time format. For information about date and date-time formats, see the *Describing Data With WebFOCUS Language* manual.

Syntax: How to Convert a Date Value to an Alphanumeric String in a Specified Date Format

```
DT_FORMAT(date,'date_format')
```

where:

date

Numeric, date, or date-time

Is the date or date-time field or value to be converted.

'*date_format*'

Alphanumeric literal

Is a date or date-time format that fits the input date format type, enclosed in single quotation marks.

Example: Converting Date and Date_Time Values to Alphanumeric Format

DT_FORMAT converts the current date and time down to the seconds to a string in date-time format HYYMTDs:

```
DT_FORMAT( DT_CURRENT_DATETIME(SECOND),'HYYMTDs')
```

On December 17, 2019 at approximately 11:36 A.M., the result is:

```
2019 December 17 11:36:45.000
```

FPRINT: Displaying a Value in a Specified Format

Given an output format, the simplified conversion function FPRINT converts a value to alphanumeric format for display.

Note: A legacy FPRINT function also exists and is still supported. For information, see *FPRINT: Converting Fields to Alphanumeric Format* on page 297. The legacy function has an additional argument for the name or format of the returned value.

Syntax: How to Display a Value in a Specified Format

```
FPRINT(value, 'out_format')
```

where:

value

Any data type

Is the value to be converted.

'*out_format*'

Fixed length alphanumeric

Is the display format. For information about valid display formats, see the manual.

Example: **Displaying a Value in a Specified Format**

FPRINT converts a date to alphanumeric format.

```
FPRINT(TIME_DATE, 'YYMtrD')
```

For 01/03/2009, the result is 2009, January 3.

HEXTYPE: Returning the Hexadecimal View of an Input Value

The HEXTYPE function returns the hexadecimal view of an input value of any data type. The result is returned as variable length alphanumeric. The alphanumeric field to which the hexidecimal value is returned must be large enough to hold two characters for each input character. The value returned depends on the running operating environment.

Syntax: **How to Returning the Hexadecimal View of an Input Value**

```
HEXTYPE(in_value)
```

where:

in_value

Is an alphanumeric or integer field, constant, or expression.

Example: **Returning a Hexadecimal View**

HEXTYPE returns a hexidecimal view of COUNTRY_NAME.

```
HEXTYPE(COUNTRY_NAME)
```

For Argentina, the result is 417267656E74696E61.

PHONETIC: Returning a Phonetic Key for a String

PHONETIC calculates a phonetic key for a string, or a null value on failure. Phonetic keys are useful for grouping alphanumeric values, such as names, that may have spelling variations. This is done by generating an index number that will be the same for the variations of the same name based on pronunciation. One of two phonetic algorithms can be used for indexing, Metaphone and Soundex. Metaphone is the default algorithm, except on z/OS where the default is Soundex.

You can set the algorithm to use with the following command.

```
SET PHONETIC_ALGORITHM = {METAPHONE|SOUNDEX}
```

Most phonetic algorithms were developed for use with the English language. Therefore, applying the rules to words in other languages may not give a meaningful result.

Metaphone is suitable for use with most English words, not just names. Metaphone algorithms are the basis for many popular spell checkers.

Note: Metaphone is not optimized in generated SQL. Therefore, if you need to optimize the request for an SQL DBMS, the SOUNDEX setting should be used.

Soundex is a legacy phonetic algorithm for indexing names by sound, as pronounced in English.

Syntax: **How to Return a Phonetic Key**

```
PHONETIC(string)
```

where:

string

Alphanumeric

Is a string for which to create the key. A null value will be returned on failure.

Example: **Generating a Phonetic Key**

PHONETIC generates a phonetic key for LAST_NAME:

```
PHONETIC(LAST_NAME)
```

For last names SMITH and SMYTHE, the same phonetic key, S530, is generated.

TO_INTEGER: Converting a Character String to an Integer Value

TO_INTEGER converts a character string that contains a valid number consisting of digits and an optional decimal point to an integer value. If the value contains a decimal point, the value after the decimal point is truncated. If the value does not represent a valid number, zero (0) is returned.

Syntax: How to Convert a Character String to an Integer

```
TO_INTEGER(string)
```

where:

string

Is a character string enclosed in single quotation marks or a character field that represents a number containing digits and an optional decimal point.

Example: Converting a Character String to an Integer Value

TO_INTEGER converts the character string '56.78' to an integer.

```
TO_INTEGER('56.78')
```

The result is 56.

TO_NUMBER: Converting a Character String to a Numeric Value

TO_NUMBER converts a character string that contains a valid number consisting of digits and an optional decimal point to the numeric format most appropriate to the context. If the value does not represent a valid number, zero (0) is returned.

Syntax: How to Convert a Character String to a Number

```
TO_NUMBER(string)
```

where:

string

Is a character string enclosed in single quotation marks or a character field that represents a number containing digits and an optional decimal point. This string will be converted to a double-precision floating point number.

Example: Converting a Character String to a Number

TO_NUMBER converts the string '56.78' to a number with one decimal place.

```
TO_NUMBER('56.78')
```

The result is 56.8.

Chapter 12

Format Conversion Functions

Format conversion functions convert fields from one format to another.

In this chapter:

ATODBL: Converting an Alphanumeric String to Double-Precision Format

The ATODBL function converts a number in alphanumeric format to decimal (double-precision) format.

Syntax: **How to Convert an Alphanumeric String to Double-Precision Format**

```
ATODBL(source_string, length, output)
```

where:

source_string
Alphanumeric

Is the string consisting of digits and, optionally, one sign and one decimal point to be converted.

length

Alphanumeric

Is the length of the source string in bytes. This can be a numeric constant, or a field or variable that contains the value. If you specify a numeric constant, enclose it in single quotation marks, for example '12'.

output

Double precision floating-point

Example: **Converting an Alphanumeric Field to Double-Precision Format**

ATODBL converts EMP_ID into double-precision format.

```
ATODBL(EMP_ID, '09', 'D12.2')
```

For 112847612, the result is 112,847,612.00.

For 117593129, the result is 117,593,129.00.

EDIT: Converting the Format of a Field

The EDIT function converts an alphanumeric field that contains numeric characters to numeric format or converts a numeric field to alphanumeric format.

This function is useful for manipulating a field in an expression that performs an operation that requires operands in a particular format.

When EDIT assigns a converted value to a new field, the format of the new field must correspond to the format of the returned value. For example, if EDIT converts a numeric field to alphanumeric format, you must give the new field an alphanumeric format:

```
DEFINE ALPHAPRICE/A6 = EDIT(PRICE);
```

EDIT deals with a symbol in the following way:

- ❑ When an alphanumeric field is converted to numeric format, a sign or decimal point in the field is stored as part of the numeric value.

  Any other non-numeric characters are invalid, and EDIT returns the value zero.

- ❑ When converting a floating-point or packed-decimal field to alphanumeric format, EDIT removes the sign, the decimal point, and any number to the right of the decimal point. It then right-justifies the remaining digits and adds leading zeros to achieve the specified field length. Converting a number with more than nine significant digits in floating-point or packed-decimal format may produce an incorrect result.

EDIT also extracts characters from or add characters to an alphanumeric string. For more information, see *EDIT: Extracting or Adding Characters* on page 122.

Syntax: How to Convert the Format of a Field

```
EDIT(fieldname);
```

where:

fieldname

Alphanumeric or Numeric

Is the field name.

Example: Converting From Numeric to Alphanumeric Format

EDIT converts HIRE_DATE (a legacy date format) to alphanumeric format.

```
EDIT(HIRE_DATE)
```

For 82/04/01, the result is 820401.

For 81/11/02, the result is 811102.

FPRINT: Converting Fields to Alphanumeric Format

The FPRINT function converts any type of field except for a text field to its alphanumeric equivalent for display. The alphanumeric representation will include any display options that are specified in the format of the original field.

Syntax: How to Convert Fields Using FPRINT

```
FPRINT(in_value, 'usageformat', output)
```

where:

in_value

Any format except TX

Is the value to be converted.

usageformat

Alphanumeric

Is the usage format of the value to be converted, including display options. The format must be enclosed in single quotation marks.

output

Alphanumeric

The output format must be long enough to hold the converted number itself, with a sign and decimal point, plus any additional characters generated by display options, such as commas, a currency symbol, or a percent sign.

For example, D12.2 format is converted to A14 because it outputs two decimal digits, a decimal point, a possible minus sign, up to eight integer digits, and two commas. If the output format is not large enough, excess right-hand characters may be truncated.

Reference: Usage Notes for the FPRINT Function

- ❏ The USAGE format must match the actual data in the field.
- ❏ The output of FPRINT for numeric values is right-justified within the area required for the maximum number of characters corresponding to the supplied format. This ensures that all possible values are aligned vertically along the decimal point or units digit.

Example: Converting a Numeric Field to Alphanumeric Format

FPRINT converts CURR_SAL (format D12.2)M to a column with format A15:

```
FPRINT(CURR_SAL, 'D12.2M', 'A15')
```

FTOA: Converting a Number to Alphanumeric Format

The FTOA function converts a number up to 16 digits long from numeric format to alphanumeric format. It retains the decimal positions of the number and right-justifies it with leading spaces. You can also add edit options to a number converted by FTOA.

When using FTOA to convert a number containing decimals to a character string, you must specify an alphanumeric format large enough to accommodate both the integer and decimal portions of the number. For example, a D12.2 format is converted to A14. If the output format is not large enough, decimals are truncated.

Syntax: How to Convert a Number to Alphanumeric Format

```
FTOA(number, '(format)', output)
```

where:

number

Numeric F or D (single and double precision floating-point)

Is the number to be converted.

format

Alphanumeric

Is the format of the number to be converted enclosed in parentheses. Only floating point single-precision and double-precision formats are supported. Include any edit options that you want to appear in the output. The D (floating-point double-precision) format automatically supplies commas.

output

Alphanumeric

The length of this argument must be greater than the length of *number* and must account for edit options and a possible negative sign.

Example: Converting From Numeric to Alphanumeric Format

FTOA converts GROSS from floating point double-precision to alphanumeric format.

```
FTOA(GROSS, '(D12.2)', 'A15')
```

For $1,815.00, the result is 1,815.00.

For $2,255.00, the result is 2,255.00.

HEXBYT: Converting a Decimal Integer to a Character

The HEXBYT function obtains the ASCII, EBCDIC, or Unicode character equivalent of a decimal integer, depending on your configuration and operating environment. The decimal value you specify must be the value associated with the character on the configured code page. HEXBYT returns a single alphanumeric character in the ASCII, EBCDIC, or Unicode character set. You can use this function to produce characters that are not on your keyboard, similar to the CTRAN function.

In Unicode configurations, this function uses values in the range:

- 0 to 255 for 1-byte characters.
- 256 to 65535 for 2-byte characters.
- 65536 to 16777215 for 3-byte characters.
- 16777216 to 4294967295 for 4-byte characters (primarily for EBCDIC).

The display of special characters depends on your software and hardware; not all special characters may appear.

Syntax: How to Convert a Decimal Integer to a Character

```
HEXBYT(decimal_value, output)
```

where:

decimal_value

Integer

Is the decimal integer to be converted to a single character. In non-Unicode environments, a value greater than 255 is treated as the remainder of *decimal_value* divided by 256. The decimal value you specify must be the value associated with the character on the configured code page.

output

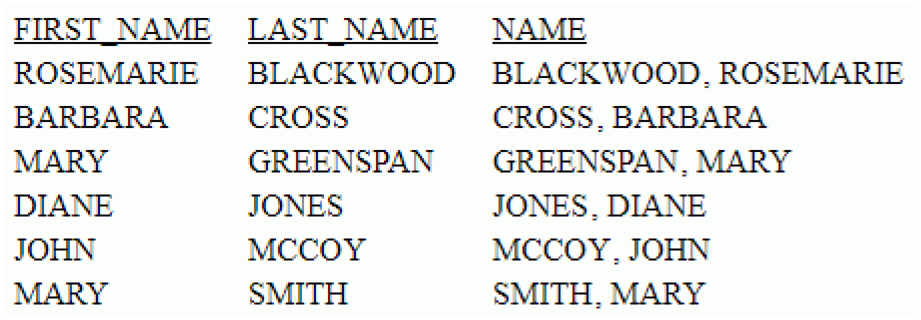
Alphanumeric

Example: Converting a Decimal Integer to a Character in ASCII and Unicode

The following request uses HEXBYT to convert the decimal integer value 130 to the comma character on ASCII code page 1252. The comma is then concatenated between LAST_NAME and FIRST_NAME to create the NAME field:

```
TABLE FILE EMPLOYEE
PRINT LAST_NAME AND
COMPUTE COMMA1/A1 = HEXBYT(130, COMMA1); NOPRINT
COMPUTE NAME/A40 = LAST_NAME || COMMA1| ' '| FIRST_NAME;
BY LAST_NAME NOPRINT
BY FIRST_NAME
WHERE DEPARTMENT EQ 'MIS';
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image.



FIRST_NAME	LAST_NAME	NAME
ROSEMARIE	BLACKWOOD	BLACKWOOD, ROSEMARIE
BARBARA	CROSS	CROSS, BARBARA
MARY	GREENSPAN	GREENSPAN, MARY
DIANE	JONES	JONES, DIANE
JOHN	MCCOY	MCCOY, JOHN
MARY	SMITH	SMITH, MARY

To produce the same output in a Unicode environment configured for code page 65001, replace the COMPUTE command for the field COMMA1 with the following syntax, in which the call to HEXBYT converts the integer value 14844058 to the comma character:

```
COMPUTE COMMA1/A1 = HEXBYT(14844058, COMMA1); NOPRINT
```

Example: Converting a Decimal Integer to a Character

HEXBYT converts LAST_INIT_CODE to its character equivalent and stores the result in a column with the format A1.

```
HEXBYT(LAST_INIT_CODE, 'A1')
```

On an ASCII platform, for 83, the result is S.

On ASCII platform, for 74, the result is J.

ITONUM: Converting a Large Number to Double-Precision Format

The ITONUM function converts a large number in a non-FOCUS data source from special long integer to double-precision format.

This is useful for some programming languages and some non-FOCUS data storage systems that use special long integers, which do not fit the regular integer format (four bytes in length) supported in the synonym, and, therefore, require conversion to double-precision format.

You must specify how many of the right-most bytes in the input field are significant. The result is an 8-byte double-precision field.

Syntax: How to Convert a Large Binary Integer to Double-Precision Format

```
ITONUM(maxbytes, infield, output)
```

where:

maxbytes

Numeric

Is the maximum number of bytes in the 8-byte binary input field that have significant numeric data, including the binary sign. Valid values are:

5 ignores the left-most 3 bytes.

6 ignores the left-most 2 bytes.

7 ignores the left-most byte.

infield

A8

Is the field that contains the number. Both the USAGE and ACTUAL formats of the field must be A8.

output

Double precision floating-point (D*n*)

Example: **Converting a Large Binary Integer to Double-Precision Format**

ITONUM converts BINARYFLD to double-precision format.

```
ITONUM(6, BINARYFLD, 'D14')
```

ITOPACK: Converting a Large Binary Integer to Packed-Decimal Format

The ITOPACK function converts a large binary integer in a non-FOCUS data source to packed-decimal format.

This is useful for some programming languages and some non-FOCUS data storage systems that use special long integers, which do not fit the regular integer format (four bytes in length) supported in the synonym, and, therefore, require conversion to packed-decimal format.

You must specify how many of the right-most bytes in the input field are significant. The result is an 8-byte packed-decimal field of up to 15 significant numeric positions (for example, P15 or P16.2).

Limit: For a field defined as 'PIC 9(15) COMP' or the equivalent (15 significant digits), the maximum number that can be converted is 167,744,242,712,576.

Syntax: **How to Convert a Large Binary Integer to Packed-Decimal Format**

```
ITOPACK(maxbytes, infield, output)
```

where:

maxbytes

Numeric

Is the maximum number of bytes in the 8-byte input field that have significant numeric data, including the binary sign.

Valid values are:

❑ **5** ignores the left-most 3 bytes (up to 11 significant positions).

- ❑ **6** ignores the left-most 2 bytes (up to 14 significant positions).
- ❑ **7** ignores the left-most byte (up to 15 significant positions).

infield

A8

Is the field that contains the binary number. Both the USAGE and ACTUAL formats of the field must be A8.

output

Numeric

The format must be P*n* or P*n*.*d*.

Example: Converting a Large Binary Integer to Packed-Decimal Format

ITOPACK converts BINARYFLD to packed-decimal format.

```
ITOPACK(6, BINARYFLD, 'P14.4')
```

ITOZ: Converting a Number to Zoned Format

The ITOZ function converts a number in numeric format to zoned-decimal format. Although a request cannot process zoned numbers, it can write zoned fields to an extract file for use by an external program.

Syntax: How to Convert a Number to Zoned Format

```
ITOZ(length, in_value, output)
```

where:

length

Integer

Is the length of *in_value* in bytes. The maximum number of bytes is 15. The last byte includes the sign.

in_value

Numeric

Is the number to be converted. The number is truncated to an integer before it is converted.

output

Alphanumeric

Example: **Converting a Number to Zoned Format**

ITOZ converts CURR_SAL to zoned format.

```
ITOZ(8, CURR_SAL, 'A8')
```

PCKOUT: Writing a Packed Number of Variable Length

The PCKOUT function writes a packed-decimal number of variable length to an extract file. When a request saves a packed number to an extract file, it typically writes it as an 8- or 16-byte field regardless of its format specification. With PCKOUT, you can vary the field's length between 1 to 16 bytes.

Syntax: **How to Write a Packed Number of Variable Length**

```
PCKOUT(in_value, length, output)
```

where:

in_value

Numeric

Is the input value. It can be in packed, integer, single- or double-precision floating point format. If it is not in integer format, it is rounded to the nearest whole number.

length

Numeric

Is the length of the output value, from 1 to 16 bytes.

output

Alphanumeric

The function returns the field as alphanumeric although it contains packed data.

Example: **Writing a Packed Number of Variable Length**

PCKOUT converts CURR_SAL to a five-byte packed format.

```
PCKOUT(CURR_SAL, 5, 'A5')
```

PTOA: Converting a Packed-Decimal Number to Alphanumeric Format

The PTOA function converts a number from numeric format to alphanumeric format. It retains the decimal positions of the number and right-justifies it with leading spaces. You can also add edit options to a number converted by PTOA.

When using PTOA to convert a number containing decimals to a character string, you must specify an alphanumeric format large enough to accommodate both the integer and decimal portions of the number. For example, a P12.2C format is converted to A14. If the output format is not large enough, the right-most characters are truncated.

Syntax: How to Convert a Packed-Decimal Number to Alphanumeric Format

```
PTOA(number, '(format)', output)
```

where:

number

Numeric P (packed-decimal) or F or D (single and double precision floating-point)

Is the number to be converted.

format

Alphanumeric

Is the format of the number enclosed in parentheses.

output

Alphanumeric

The length of this argument must be greater than the length of *number* and must account for edit options and a possible negative sign.

Example: Converting From Packed to Alphanumeric Format

PTOA converts PGROSS from packed-decimal to alphanumeric format.

```
PTOA(PGROSS, FMT, 'A17')
```

TSTOPACK: Converting an MSSQL or Sybase Timestamp Column to Packed Decimal

This function applies to the Microsoft SQL Server and Sybase adapters only.

Microsoft SQL Server and Sybase have a data type called TIMESTAMP. Rather than containing an actual timestamp, columns with this data type contain a number that is incremented for each record inserted or updated in the data source. This timestamp comes from a common area, so no two tables in the database have the same timestamp column value. The value is stored in Binary(8) or Varbinary(8) format in the table, but is returned as a double wide alphanumeric column (A16). You can use the TSTOPACK function to convert the timestamp value to packed decimal.

Syntax: How to Convert an MSSQL or Sybase Timestamp Column to Packed Decimal

```
TSTOPACK(tscol, output);
```

where:

tscol

A16

Is the timestamp column to be converted.

output

P21

Example: Converting a Microsoft SQL Server Timestamp Column to Packed Decimal

The Master File for the TSTEST data source follows. The field TS represents the TIMESTAMP column:

```
FILENAME=TSTEST, SUFFIX=SQLMSS  , $
  SEGMENT=TSTEST, SEGTYPE=S0, $
    FIELDNAME=I, ALIAS=I, USAGE=I11, ACTUAL=I4,
      MISSING=ON, $
    FIELDNAME=TS, ALIAS=TS, USAGE=A16, ACTUAL=A16, FIELDTYPE=R, $
```

Note: When you generate a synonym for a table with a TIMESTAMP column, the TIMESTAMP column is created as read-only (FIELDTYPE=R).

TSTOPACK converts the timestamp column TS to packed decimal:

```
TSNUM/P21=TSTOPACK(TS,'P21');
```

For 0000000000007815, the result is 30741.

For 0000000000007816, the result is 30742.

UFMT: Converting an Alphanumeric String to Hexadecimal

The UFMT function converts characters in an alphanumeric source string to their hexadecimal representation. This function is useful for examining data of unknown format. As long as you know the length of the data, you can examine its content.

Syntax: How to Convert an Alphanumeric String to Hexadecimal

```
UFMT(source_string, length, output)
```

where:

source_string

Alphanumeric

Is the alphanumeric string to convert.

length

Integer

Is the number of characters in *source_string*.

output

Alphanumeric

The format of *output* must be alphanumeric and its length must be twice that of *length*.

Example: Converting an Alphanumeric String to Hexadecimal

UFMT converts each value in JOBCODE to its hexadecimal representation and stores it in a column with the format A6.

```
UFMT(JOBCODE, 3, 'A6')
```

For A01, the result is C1F0F1.

For A02, the result is C1F0F2.

XTPACK: Writing a Packed Number With Up to 31 Significant Digits to an Output File

The XTPACK function stores packed numbers with up to 31 significant digits in an alphanumeric field, retaining decimal data. This permits writing a short or long packed field of any length, 1 to 16 bytes, to an output file.

Syntax: How to Store Packed Values in an Alphanumeric Field

```
XTPACK(in_value, outlength, outdec, output)
```

where:

infield

Numeric

Is the packed value.

outlength

Numeric

Is the length of the alphanumeric field that will hold the converted packed field. Can be from 1 to 16.

outdec

Numeric

Is the number of decimal positions for *output*.

output

Alphanumeric

Example: Writing a Long Packed Number to an Output File

XTPACK converts LONGPCK to alphanumeric so that it can be saved in an output file:

```
XTPACK(LONGPCK,13,2,'A13');
```

Chapter 13

Simplified Numeric Functions

Numeric functions have been developed that make it easier to understand and enter the required arguments. These functions have streamlined parameter lists, similar to those used by SQL functions. In some cases, these simplified functions provide slightly different functionality than previous versions of similar functions.

The simplified functions do not have an output argument. Each function returns a value that has a specific data type.

When used in a request against a relational data source, these functions are optimized (passed to the RDBMS for processing).

Note:

❑ The simplified numeric functions are supported in Dialogue Manager.

In this chapter:

CEILING: Returning the Smallest Integer Value Greater Than or Equal to a Value

CEILING returns the smallest integer value that is greater than or equal to a number.

Syntax: **How to Return the Smallest Integer Greater Than or Equal to a Number**

```
CEILING(number)
```

where:

number

Numeric

Is the number whose ceiling will be returned. The output data type is the same as the input data type.

Example: **Returning the Ceiling of a Number**

CEILING returns the smallest integer larger than the value in GROSS_PROFIT_US:

```
CEILING(GROSS_PROFIT_US)
```

For *225.98*, the output is *226.00*.

For *-30.01*, the output is *-30.00*.

EXPONENT: Raising e to a Power

EXPONENT raises the constant *e* to a power.

Syntax: **How to Raise the Constant e to a Power**

```
EXPONENT(power)
```

where:

power

Numeric

Is the power to which to raise e. The output data type is numeric.

Example: **Raising e to a Power**

For EXPONENT(1), the value is 2.71828

For EXPONENT(5), the value is 148.41316

FLOOR: Returning the Largest Integer Less Than or Equal to a Value

FLOOR returns the largest integer value that is less than or equal to a number.

Syntax: **How to Return the Largest Integer Less Than or Equal to a Number**

```
FLOOR(number)
```

where:

number

Numeric

Is the number whose floor will be returned. The output data type is the same as the input data type.

Example: **Returning the Floor of a Number**

FLOOR returns the smallest integer larger than the value in GROSS_PROFIT_US:

```
FLOOR(GROSS_PROFIT_US)
```

For *225.98*, the output is *225.00*.

For *-30.01*, the output is *-31.00*.

LOG10: Calculating the Base 10 Logarithm

LOG10 returns the base-10 logarithm of a numeric expression.

Syntax: **How to Calculate the Base 10 Logarithm**

```
LOG10(num_exp)
```

where:

num_exp

Numeric

Is the numeric value for which to calculate the base 10 logarithm.

Example: **Calculating the Base 10 Logarithm**

LOG10 calculates the base 10 log of NUMBER.

```
LOG10(NUMBER)
```

For 145, the result is 2.161.

MOD: Calculating the Remainder From a Division

MOD calculates the remainder from a division. The output data type is the same as the input data type.

Syntax: **How to Calculate the Remainder From a Division**

```
MOD(dividend, divisor)
```

where:

`dividend`

Numeric

Is the value to divide.

Note: The sign of the returned value will be the same as the sign of the dividend.

`divisor`

Numeric

Is the value to divide by.

If the divisor is zero (0), the dividend is returned.

Example: Calculating the Remainder From a Division

MOD returns the remainder of PRICE_DOLLARS divided by DAYSDELAYED

```
MOD(PRICE_DOLLARS, DAYSDELAYED)
```

For *399.00/3*, the value is zero (0).

for *489.00/3*, the value is .99.

POWER: Raising a Value to a Power

POWER raises a base value to a power.

Syntax: How to Raise a Value to a Power

```
POWER(base, power)
```

where:

`base`

Numeric

Is the value to raise to a power. The output value has the same data type as the base value. If the base value is integer, negative power values will result in truncation.

`power`

Numeric

Is the power to which to raise the base value.

Example: Raising a Base Value to a Power

Power returns the value COGS_US/20.00 raised to the power stored in DAYSDELAYED.

```
POWER1= POWER(COGS_US/20.00,DAYSDELAYED)
```

For base 12.15 and power 3, the value is 1,793.61

ROUND: Rounding a Number to a Given Number of Decimal Places

Given a numeric expression and an integer count, ROUND returns the numeric expression rounded to that number of decimal places. If the number of decimal places is negative, it rounds to the left of the decimal point.

Syntax: How to Round a Number to a Given Number of Decimal Places

```
ROUND(num_exp, count)
```

where:

num_exp

Numeric

Is the numeric expression to be rounded.

count

Numeric

Is the number of decimal places to which the numeric expression is to be rounded. If the number of decimal places is negative, ROUND rounds to the left of the decimal point.

Example: Rounding a Number to a Given Number of Decimal Places

ROUND rounds the number 1234.56 to -3 decimal places.

```
ROUND(1.23456, 3)
```

The result is 1.23500.

ROUND rounds the number 1.23456 to 3 decimal places.

```
ROUND(1234.56, -3)
```

The result is 1000.00.

SIGN: Returning the Sign of a Number

SIGN takes a numeric argument and returns the value -1 if the number is negative, 0 (zero) if the number is zero, and 1 if the number is positive.

Syntax: How to Return the Sign of a Number

```
SIGN(number)
```

where:

number

Is a field containing a numeric value or a number.

Example: Returning the Sign of a Number

SIGN(-5.5) returns -1.

SIGN(4) returns 1.

SIGN(0) returns 0.

TRUNCATE: Truncating a Number to a Given Number of Decimal Places

Given a numeric expression and an integer count, TRUNCATE returns the numeric expression truncated to that number of decimal places. If the number of decimal places is negative, it truncates to the left of the decimal point.

Syntax: How to Truncate a Number to a Given Number of Decimal Places

```
TRUNCATE(num_exp, count)
```

where:

num_exp

Numeric

Is the numeric expression to be truncated.

count

Numeric

Is the number of decimal places to which the numeric expression is to be truncated. If the number of decimal places is negative, TRUNCATE truncates to the left of the decimal point.

Example: Truncating a Number to a Given Number of Decimal Places

TRUNCATE truncates 1.23456 to 3 decimal places.

```
TRUNCATE(1.23456, 3)
```

The result is 1.23400.

Chapter 14 Numeric Functions

Numeric functions perform calculations on numeric constants and fields.

Note: With CDN ON, numeric arguments must be delimited by a comma followed by a space.

In this chapter:

ABS: Calculating Absolute Value

The ABS function returns the absolute value of a number.

Syntax: How to Calculate Absolute Value

```
ABS(in_value)
```

where:

`in_value`

Numeric

Is the value for which the absolute value is returned. If you use an expression, use parentheses as needed to ensure the correct order of evaluation.

Example: Calculating Absolute Value

ABS calculates the absolute value of DIFF.

```
ABS(DIFF);
```

For 15, the result is 15.

For -2, the result is 2.

CHKPCK: Validating a Packed Field

The CHKPCK function validates the data in a field described as packed format (if available on your platform). The function prevents a data exception from occurring when a request reads a field that is expected to contain a valid packed number but does not.

To use CHKPCK:

1. Ensure that the Master File (USAGE and ACTUAL attributes) defines the field as alphanumeric, not packed. This does *not* change the field data, which remains packed, but it enables the request to read the data without a data exception.
2. Call CHKPCK to examine the field. The function returns the output to a field defined as packed. If the value it examines is a valid packed number, the function returns the value; if the value is not packed, the function returns an error code.

Syntax: How to Validate a Packed Field

```
CHKPCK(length, in_value, error, output)
```

where:

`length`

Numeric

Is the number of bytes in the packed field. It can be between 1 and 16 bytes.

infield

Alphanumeric

Is the value to be verified as packed decimal. The value must be described as alphanumeric, not packed.

error

Numeric

Is the error code that the function returns if a value is not packed. Choose an error code outside the range of data. The error code is first truncated to an integer, then converted to packed format. However, it may appear on a report with a decimal point depending on the output format.

output

Packed-decimal

Example: **Validating Packed Data**

CHKPCK validates the values in PACK_SAL, and store the result in a column with the format P8CM. Values not in packed format return the error code -999. Values in packed format appear accurately.

```
CHKPCK(8, PACK_SAL, -999, 'P8CM')
```

DMOD, FMOD, and IMOD: Calculating the Remainder From a Division

The MOD functions calculate the remainder from a division. Each function returns the remainder in a different format.

The functions use the following formula.

```
remainder = dividend - INT(dividend/divisor) * divisor
```

- *DMOD* returns the remainder as a decimal number.
- *FMOD* returns the remainder as a floating-point number.
- *IMOD* returns the remainder as an integer.

For information on the INT function, see *INT: Finding the Greatest Integer* on page 322.

Syntax: **How to Calculate the Remainder From a Division**

```
function(dividend, divisor, output)
```

where:

function

Is one of the following:

DMOD returns the remainder as a decimal number.

FMOD returns the remainder as a floating-point number.

IMOD returns the remainder as an integer.

dividend

Numeric

Is the number being divided.

divisor

Numeric

Is the number dividing the dividend.

output

Numeric

Is the result whose format is determined by the function used.

If the divisor is zero (0), the dividend is returned.

Example: **Calculating the Remainder From a Division**

IMOD divides ACCTNUMBER by 1000 and stores the remainder in a column with the format I3L.

```
IMOD(ACCTNUMBER, 1000, 'I3L')
```

For 122850108, the result is 108.

For 163800144, the result is 144.

EXP: Raising *e* to the Nth Power

The EXP function raises the value "e" (approximately 2.72) to a specified power. This function is the inverse of the LOG function, which returns the logarithm of the argument.

EXP calculates the result by adding terms of an infinite series. If a term adds less than .000001 percent to the sum, the function ends the calculation and returns the result as a double-precision number.

Syntax: How to Raise *e* to the Nth Power

```
EXP(power, output)
```

where:

power

Numeric

Is the power to which "e" is raised.

output

Double-precision floating-point

Example: Raising *e* to the Nth Power

EXP raises "e" to the power designated by the &POW variable, specified here as 3. The result is then rounded to the nearest integer with the .5 rounding constant. The result has the format D15.3.

```
EXP(&POW, 'D15.3') + 0.5;
```

For 3, the result is APPROXIMATELY 20.

EXPN: Evaluating a Number in Scientific Notation

The EXPN function evaluates a numeric literal or Dialogue Manager variable expressed in scientific notation.

Syntax: How to Evaluate a Number in Scientific Notation

```
EXPN(n.nn {E|D} {+|-} p)
```

where:

n.nn

Numeric

Is a numeric literal that consists of a whole number component, followed by a decimal point, followed by a fractional component.

`E, D`

Denotes scientific notation. E and D are interchangeable.

`+, -`

Indicates if *p* is positive or negative.

p

Integer

Is the power of 10 to which to raise *n.nn*.

Note: EXPN does not use an output argument. The format of the result is floating-point double precision.

Example: Evaluating a Number in Scientific Notation

EXPN evaluates 1.03E+2.

```
EXPN(1.03E+2)
```

The result is 103.

INT: Finding the Greatest Integer

The INT function returns the integer component of a number.

Syntax: How to Find the Greatest Integer

```
INT(in_value)
```

where:

in_value

Numeric

Is the value for which the integer component is returned. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation.

Note: INT does not use an output argument. The format of the result is floating-point double precision.

Example: **Finding the Greatest Integer**

INT finds the greatest integer in DED_AMT.

```
INT(DED_AMT)
```

For $1,261.40, the result is 1261.

For $1,668.69, the result is 1668.

LOG: Calculating the Natural Logarithm

The LOG function returns the natural logarithm of a number.

Syntax: **How to Calculate the Natural Logarithm**

```
LOG(in_value)
```

where:

in_value

Numeric

Is the value for which the natural logarithm is calculated. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation. If *in_value* is less than or equal to 0, LOG returns 0.

Note: LOG does not use an output argument. The format of the result is floating-point double precision.

Example: **Calculating the Natural Logarithm**

LOG calculates the logarithm of CURR_SAL.

```
LOG(CURR_SAL)
```

For $29,700.00, the result is 10.30.

For $26,862.00, the result is 10.20.

MAX and MIN: Finding the Maximum or Minimum Value

The MAX and MIN functions return the maximum or minimum value, respectively, from a list of values.

Syntax: **How to Find the Maximum or Minimum Value**

```
{MAX|MIN}(value1, value2, ...)
```

where:

`MAX`

Returns the maximum value.

`MIN`

Returns the minimum value.

`value1, value2`

Numeric

Are the values for which the maximum or minimum value is returned. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation.

Note: MAX and MIN do not use an output argument. The format of the result is floating-point double precision.

Example: **Determining the Minimum Value**

MIN returns either the value of ED_HRS or the constant 30, whichever is lower.

```
MIN(ED_HRS, 30)
```

For 45.00, the result is 30.00.

For 25.00, the result is 25.00.

NORMSDST and NORMSINV: Calculating Normal Distributions

The NORMSDST and NORMSINV functions perform calculations on a standard normal distribution curve. NORMSDST calculates the percentage of data values that are less than or equal to a normalized value; NORMSINV is the inverse of NORMSDST, calculates the normalized value that forms the upper boundary of a percentile in a standard normal distribution curve.

NORMSDST: Calculating Standard Cumulative Normal Distribution

The NORMSDST function performs calculations on a standard normal distribution curve, calculating the percentage of data values that are less than or equal to a normalized value. A normalized value is a point on the X-axis of a standard normal distribution curve in standard deviations from the mean. This is useful for determining percentiles in normally distributed data.

The NORMSINV function is the inverse of NORMSDST. For information about NORMSINV, see *NORMSINV: Calculating Inverse Cumulative Normal Distribution* on page 327.

The results of NORMSDST are returned as double-precision and are accurate to 6 significant digits.

A standard normal distribution curve is a normal distribution that has a mean of 0 and a standard deviation of 1. The total area under this curve is 1. A point on the X-axis of the standard normal distribution is called a normalized value. Assuming that your data is normally distributed, you can convert a data point to a normalized value to find the percentage of scores that are less than or equal to the raw score.

You can convert a value (raw score) from your normally distributed data to the equivalent normalized value (z-score) as follows:

```
z = (raw_score - mean)/standard_deviation
```

To convert from a z-score back to a raw score, use the following formula:

```
raw_score = z * standard_deviation + mean
```

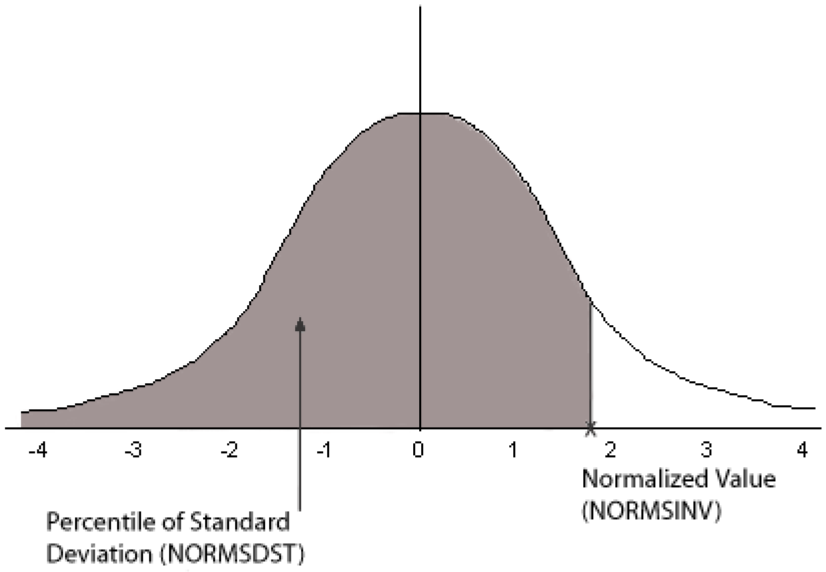
The mean of data points xi, where i is from 1 to n is:

$(\sum x_i)/n$

The standard deviation of data points xi, where i is from 1 to n is:

$\mathrm{SQRT}(\ ((\sum x_i^2 - (\sum x_i)^2/n)/(n - 1)))$

The following diagram illustrates the results of the NORMSDST and NORMSINV functions.



Reference: **Characteristics of the Normal Distribution**

Many common measurements are normally distributed. A plot of normally distributed data values approximates a bell-shaped curve. The two measures required to describe any normal distribution are the mean and the standard deviation:

- The mean is the point at the center of the curve.
- The standard deviation describes the spread of the curve. It is the distance from the mean to the point of inflection (where the curve changes direction).

Syntax: **How to Calculate the Cumulative Standard Normal Distribution Function**

```
NORMSDST(value, 'D8');
```

where:

value

Is a normalized value.

`D8`

Is the required format for the result. The value returned by the function is double-precision. You can assign it to a field with any valid numeric format.

Example: Using the NORMSDST Function

NORMSDST finds the percentile for Z and stores the result in a column with the format D8.

```
NORMSDST(Z, 'D8')
```

For -.07298, the result is .47091.

For -.80273 the result is .21106.

NORMSINV: Calculating Inverse Cumulative Normal Distribution

The NORMSINV function performs calculations on a standard normal distribution curve, finding the normalized value that forms the upper boundary of a percentile in a standard normal distribution curve. This is the inverse of NORMSDST. For information about NORMSDST, see *NORMSDST: Calculating Standard Cumulative Normal Distribution* on page 324.

The results of NORMSINV are returned as double-precision and are accurate to 6 significant digits.

Syntax: How to Calculate the Inverse Cumulative Standard Normal Distribution Function

```
NORMSINV(value, 'D8');
```

where:

`value`

Is a number between 0 and 1 (which represents a percentile in a standard normal distribution).

`D8`

Is the required format for the result. The value returned by the function is double-precision. You can assign it to a field with any valid numeric format.

Example: Using the NORMSINV Function

NORMSINV returns a normalized value from a percentile found using NORMSDST.

```
NORMSINV(NORMSD, 'D8')
```

For .21106, the result is -.80273.

For .47091, the result is -.07298

PRDNOR and PRDUNI: Generating Reproducible Random Numbers

The PRDNOR and PRDUNI functions generate reproducible random numbers:

- PRDNOR generates reproducible double-precision random numbers normally distributed with an arithmetic mean of 0 and a standard deviation of 1.
- PRDUNI generates reproducible double-precision random numbers uniformly distributed between 0 and 1 (that is, any random number it generates has an equal probability of being anywhere between 0 and 1).

Syntax: How to Generate Reproducible Random Numbers

```
{PRDNOR|PRDUNI}(seed, output)
```

where:

`PRDNOR`

Generates reproducible double-precision random numbers normally distributed with an arithmetic mean of 0 and a standard deviation of 1.

`PRDUNI`

Generates reproducible double-precision random numbers uniformly distributed between 0 and 1.

`seed`

Numeric

Is the seed or the field that contains the seed, up to 9 digits. The seed is truncated to an integer.

`output`

Double-precision

Example: Generating Reproducible Random Numbers

PRDNOR assigns random numbers and stores them in a column with the format D12.2.

```
PRDNOR(40, 'D12.2')
```

RDNORM and RDUNIF: Generating Random Numbers

The RDNORM and RDUNIF functions generate random numbers:

- RDNORM generates double-precision random numbers normally distributed with an arithmetic mean of 0 and a standard deviation of 1.
- RDUNIF generates double-precision random numbers uniformly distributed between 0 and 1 (that is, any random number it generates has an equal probability of being anywhere between 0 and 1).

Syntax: **How to Generate Random Numbers**

```
{RDNORM|RDUNIF}(output)
```

where:

`RDNORM`

Generates double-precision random numbers normally distributed with an arithmetic mean of 0 and a standard deviation of 1.

`RDUNIF`

Generates double-precision random numbers uniformly distributed between 0 and 1.

`output`

Double-precision

Example: **Generating Random Numbers**

RDNORM assigns random numbers and stores them in a column with the format D12.2.

```
RDNORM('D12.2')
```

SQRT: Calculating the Square Root

The SQRT function calculates the square root of a number.

Syntax: **How to Calculate the Square Root**

```
SQRT(in_value)
```

where:

in_value

Numeric

Is the value for which the square root is calculated. If you supply an expression, use parentheses as needed to ensure the correct order of evaluation. If you supply a negative number, the result is zero.

Note: SQRT does not use an output argument. The result of the function is floating-point double precision.

Example: **Calculating the Square Root**

SQRT calculates the square root of LISTPR.

```
SQRT(LISTPR)
```

For 19.98, the result is 4.47.

For 14.98, the result is 3.87.

Chapter 15

Simplified Statistical Functions

Simplified statistical functions can be called in a COMPUTE command to perform statistical calculations on the internal matrix that is generated during TABLE request processing. The STDDEV and CORRELATION functions can also be called as a verb object in a display command. Prior to calling a statistical function, you need to establish the size of the partition on which these functions will operate, if the request contains sort fields.

Note: It is recommended that all numbers and fields used as parameters to these functions be double-precision.

In this chapter:

Specify the Partition Size for Simplified Statistical Functions

```
SET PARTITION_ON = {FIRST|PENULTIMATE|TABLE}
```

where:

`FIRST`

Uses the first (also called the major) sort field in the request to partition the values.

`PENULTIMATE`

Uses the next to last sort field where the COMPUTE is evaluated to partition the values. This is the default value.

`TABLE`

Uses the entire internal matrix to calculate the statistical function.

CORRELATION: Calculating the Degree of Correlation Between Two Sets of Data

The CORRELATION function calculates the correlation coefficient between two numeric fields. The function returns a numeric value between zero (-1.0) and 1.0.

Syntax: How to Calculate the Correlation Coefficient Between Two Fields

```
CORRELATION(field1, field2)
```

where:

`field1`

Numeric

Is the first set of data for the correlation.

`field2`

Numeric

Is the second set of data for the correlation.

Note: Arguments for CORRELATION cannot be prefixed fields. If you need to work with fields that have a prefix operator applied, apply the prefix operators to the fields in COMPUTE commands and save the results in a HOLD file. Then, run the correlation against the HOLD file.

Example: Calculating a Correlation

CORRELATION calculates the correlation between DOLLARS and BUDDOLLARS.

```
CORRELATION(DOLLARS, BUDDOLLARS)
```

For DOLLARS=46,156,290.00 and BUDDOLLARS=46,220,778.00, the result is 0.895691073.

KMEANS_CLUSTER: Partitioning Observations Into Clusters Based on the Nearest Mean Value

The KMEANS_CLUSTER function partitions observations into a specified number of clusters based on the nearest mean value. The function returns the cluster number assigned to the field value passed as a parameter.

Note: If there are not enough points to create the number of clusters requested, the value -10 is returned for any cluster that cannot be created.

Syntax: How to Partition Observations Into Clusters Based on the Nearest Mean Value

```
KMEANS_CLUSTER(number, percent, iterations, tolerance,
       [prefix1.]field1[, [prefix1.]field2 ...])
```

where:

number

Integer

Is number of clusters to extract.

percent

Numeric

Is the percent of training set size (the percent of the total data to use in the calculations). The default value is AUTO, which uses the internal default percent.

iterations

Integer

Is the maximum number of times to recalculate using the means previously generated. The default value is AUTO, which uses the internal default number of iterations.

tolerance

Numeric

Is a weight value between zero (0) and 1.0. The value AUTO uses the internal default tolerance.

prefix1, prefix2

Defines an optional aggregation operator to apply to the field before using it in the calculation. Valid operators are:

- **SUM.** which calculates the sum of the field values. SUM is the default value.
- **CNT.** which calculates a count of the field values.
- **AVE.** which calculates the average of the field values.
- **MIN.** which calculates the minimum of the field values.
- **MAX.** which calculates the maximum of the field values.
- **FST.** which retrieves the first value of the field.
- **LST.** which retrieves the last value of the field.

Note: The operators PCT., RPCT., TOT., MDN., MDE., RNK., and DST. are not supported.

field1

Numeric

Is the set of data to be analyzed.

field2

Numeric

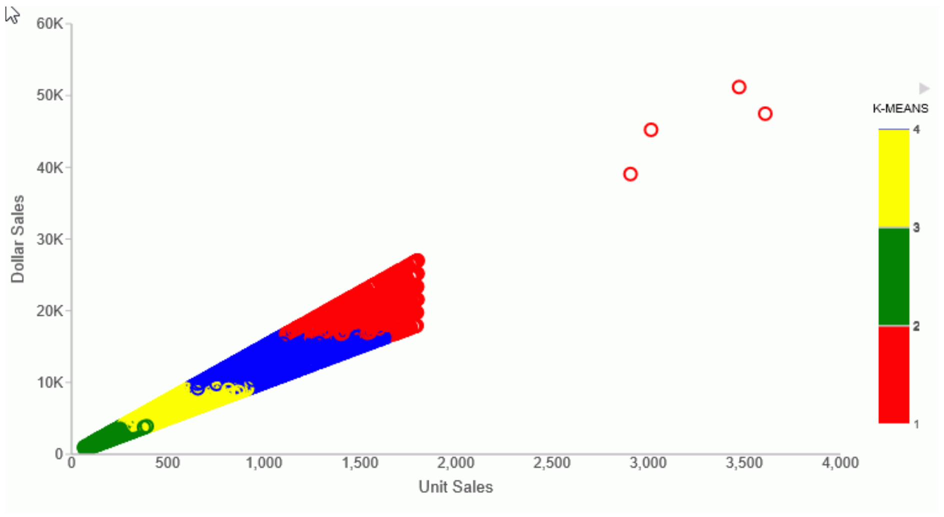
Is an optional set of data to be analyzed.

Example: Partitioning Data Values Into Clusters

The following request partitions the DOLLARS field values into four clusters and displays the result as a scatter chart in which the color represents the cluster. The request uses the default values for the percent, iterations, and tolerance parameters by passing them as the value 0 (zero).

```
SET PARTITION_ON = PENULTIMATE
GRAPH FILE GGSALES
PRINT UNITS DOLLARS
COMPUTE KMEAN1/D20.2 TITLE 'K-MEANS'=  KMEANS_CLUSTER(4, AUTO, AUTO, AUTO,
DOLLARS);
ON GRAPH SET LOOKGRAPH SCATTER
ON GRAPH PCHOLD FORMAT JSCHART
ON GRAPH SET STYLE *
INCLUDE=Warm.sty,$
type = data, column = N2, bucket=y-axis,$
type=data, column= N1, bucket=x-axis,$
type=data, column=N3, bucket=color,$
GRID=OFF,$
*GRAPH_JS_FINAL
colorScale: {
        colorMode: 'discrete',
        colorBands: [{start: 1, stop: 1.99, color: 'red'}, {start: 2, stop:
2.99, color: 'green'},
               {start: 3, stop: 3.99, color: 'yellow'}, {start: 3.99, stop:
4, color: 'blue'} ]
    }
*END
ENDSTYLE
END
```

The output is shown in the following image.



MULTIREGRESS: Creating a Multivariate Linear Regression Column

MULTIREGRESS derives a linear equation that best fits a set of numeric data points, and uses this equation to create a new column in the report output. The equation can be based on one or more independent variables.

The equation generated is of the following form, where y is the dependent variable and x1, x2, and x3 are the independent variables.

```
y = a1*x1 [+ a2*x2 [+ a3*x3] ...] + b
```

When there is one independent variable, the equation represents a straight line. When there are two independent variables, the equation represents a plane, and with three independent variables, it represents a hyperplane. You should use this technique when you have reason to believe that the dependent variable can be approximated by a linear combination of the independent variables.

Syntax: How to Create a Multivariate Linear Regression Column

```
MULTIREGRESS(input_field1, [input_field2, ...])
```

where:

input_field1, input_field2 ...

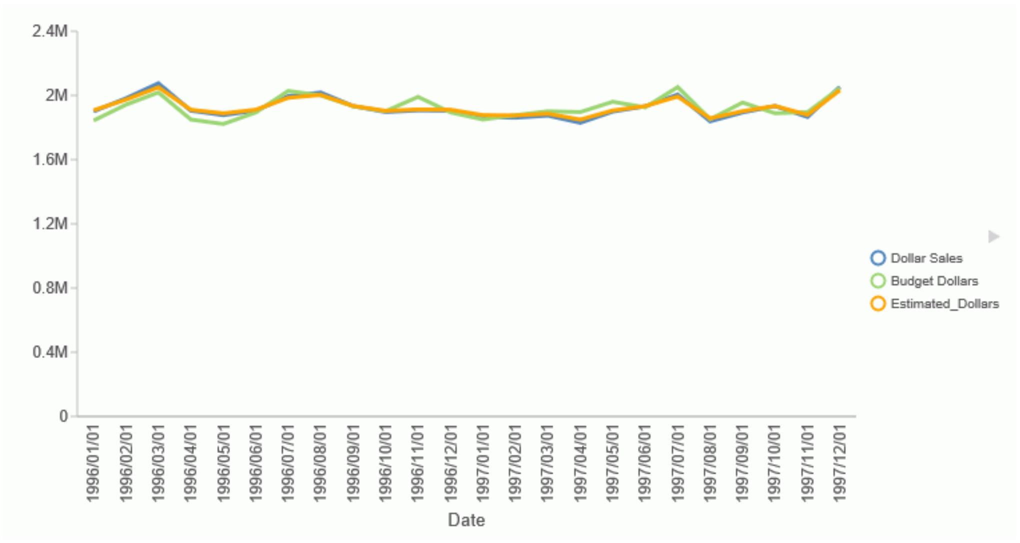
Are any number of field names to be used as the independent variables. They should be independent of each other. If an input field is non-numeric, it will be categorized to transform it to numeric values that can be used in the linear regression calculation.

Example: Creating a Multivariate Linear Regression Column

The following request uses the DOLLARS and BUDDOLLARS fields to generate a regression column named Estimated_Dollars.

```
GRAPH FILE GGSALES
SUM BUDUNITS UNITS BUDDOLLARS DOLLARS
COMPUTE Estimated_Dollars/F8 = MULTIREGRESS(DOLLARS, BUDDOLLARS);
BY DATE
ON GRAPH SET LOOKGRAPH LINE
ON GRAPH PCHOLD FORMAT JSCHART
ON GRAPH SET STYLE *
INCLUDE=Warm.sty,$
type=data, column = n1, bucket = x-axis,$
type=data, column= dollars, bucket=y-axis,$
type=data, column= buddollars, bucket=y-axis,$
type=data, column= Estimated_Dollars, bucket=y-axis,$
*GRAPH_JS
"series":[
{"series":2, "color":"orange"}]
*END
ENDSTYLE
END
```

The output is shown in the following image. The orange line represents the regression equation.



OUTLIER: Identifying Outliers in Numeric Data

The 1.5 * IQR (Inner Quartile Range) rule is a common way to identify outliers in data. This rule defines an outlier as a value that is above or below 1.5 times the inner quartile range in the data. The inner quartile range is based on sorting the data values, dividing it into equal quarters, and calculating the range of values between the first quartile (the value one quarter of the way through the sorted data) and third quartile (the value three quarters of the way through the sorted data). The value that is 1.5 times below the inner quartile range is called the *lower fence*, and the value that is 1.5 times above the inner quartile range is called the *upper fence*.

OUTLIER is not supported in a DEFINE expression. It can be used in a COMPUTE expression or a WHERE, WHERE TOTAL, or WHERE_GROUPED phrase.

Given a numeric field as input, OUTLIER returns one of the following values for each value of the field, using the 1.5 * IQR rule:

- **0 (zero).** The value is not an outlier.
- **-1.** The value is below the lower fence.
- **1.** The value is above the upper fence.

Syntax: How to Identify Outliers in Numeric Data

```
OUTLIER(input_field)
```

where:

input_field

Numeric

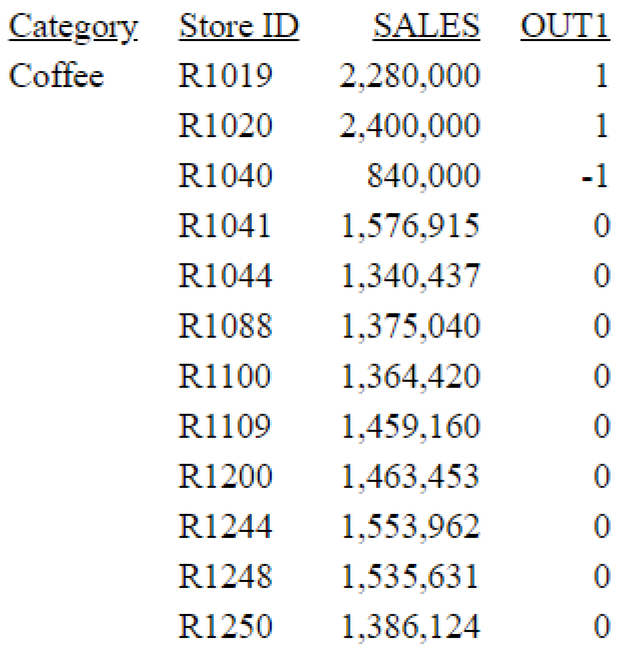
Is the numeric field to be analyzed.

Example: Identifying Outliers

The following request defines the SALES field to have different values depending on the store code, and uses OUTLIER to determine whether each field value is an outlier.

```
DEFINE FILE GGSALES
SALES/D12 = IF ((CATEGORY EQ 'Coffee') AND (STCD EQ 'R1019')) THEN 19000
  ELSE IF ((CATEGORY EQ 'Coffee') AND (STCD EQ 'R1020')) THEN 20000
  ELSE IF ((CATEGORY EQ 'Coffee') AND (STCD EQ 'R1040')) THEN 7000
  ELSE DOLLARS;
END
TABLE FILE GGSALES
SUM SALES
COMPUTE OUT1/I3 = OUTLIER(SALES);
BY CATEGORY
BY STCD
WHERE CATEGORY EQ 'Coffee'
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image. Values above 2 million are above the upper fence, values below 1 million are below the lower fence, and other values are not outliers:



Category	Store ID	SALES	OUT1
Coffee	R1019	2,280,000	1
	R1020	2,400,000	1
	R1040	840,000	-1
	R1041	1,576,915	0
	R1044	1,340,437	0
	R1088	1,375,040	0
	R1100	1,364,420	0
	R1109	1,459,160	0
	R1200	1,463,453	0
	R1244	1,553,962	0
	R1248	1,535,631	0
	R1250	1,386,124	0

RSERVE: Running an R Script

You can use the RSERVE function in a COMPUTE command to run an R script that returns vector output. This requires that you have a configured Adapter for Rserve.

Syntax: **How to Run an R Script**

```
RSERVE(rserve_mf, input_field1, ...input_fieldn, output)
```

where:

rserve_mf

Is the synonym for the R script.

input_field1, ...input_fieldn

Are the independent variables used by the R script.

output

Is the dependent variable returned by the R script. It must be a single column (vector) of output.

Example: Using RSERVE to Run an R Script

The R script named wine_run_model.R predicts Bordeaux wine prices based on the average growing season temperature, the amount of rain during the harvest season, the amount of rain during the winter, and the age of the wine.

Using a configured connection (named MyRserve) for the Adapter for Rserve, and a sample data file named wine_input_sample.csv, you create the following synonym for the R script, as described in the *Adapter Administration* manual.

Master File

```
FILENAME=WINE_RUN_MODEL, SUFFIX=RSERVE  , $
  SEGMENT=INPUT_DATA, SEGTYPE=S0, $
    FIELDNAME=AGST, ALIAS=AGST, USAGE=D9.4, ACTUAL=STRING,
      MISSING=ON,
      TITLE='AGST', $
    FIELDNAME=HARVESTRAIN, ALIAS=HarvestRain, USAGE=I11, ACTUAL=STRING,
      MISSING=ON,
      TITLE='HarvestRain', $
    FIELDNAME=WINTERRAIN, ALIAS=WinterRain, USAGE=I11, ACTUAL=STRING,
      MISSING=ON,
      TITLE='WinterRain', $
    FIELDNAME=AGE, ALIAS=Age, USAGE=I11, ACTUAL=STRING,
      MISSING=ON,
      TITLE='Age', $
  SEGMENT=OUTPUT_DATA, SEGTYPE=U, PARENT=INPUT_DATA, $
    FIELDNAME=PRICE, ALIAS=Price, USAGE=D18.14, ACTUAL=STRING,
      MISSING=ON,
      TITLE='Price', $
```

Access File

```
SEGNAME=INPUT_DATA,
  CONNECTION=MyRserve,
  R_SCRIPT=/prediction/wine_run_model.r,
  R_SCRIPT_LOCATION=WFRS,
  R_INPUT_SAMPLE_DAT=prediction/wine_input_sample.csv, $
```

Now that the synonym has been created for the model, the model will be used to run against the following data file named wine_forecast.csv.

```
Year,Price,WinterRain,AGST,HarvestRain,Age,FrancePop
1952,7.495,600,17.1167,160,31,43183.569
1953,8.0393,690,16.7333,80,30,43495.03
1955,7.6858,502,17.15,130,28,44217.857
1957,6.9845,420,16.1333,110,26,45152.252
1958,6.7772,582,16.4167,187,25,45653.805
1959,8.0757,485,17.4833,187,24,46128.638
1960,6.5188,763,16.4167,290,23,46583.995
1961,8.4937,830,17.3333,38,22,47128.005
1962,7.388,697,16.3,52,21,48088.673
1963,6.7127,608,15.7167,155,20,48798.99
1964,7.3094,402,17.2667,96,19,49356.943
1965,6.2518,602,15.3667,267,18,49801.821
1966,7.7443,819,16.5333,86,17,50254.966
1967,6.8398,714,16.2333,118,16,50650.406
1968,6.2435,610,16.2,292,15,51034.413
1969,6.3459,575,16.55,244,14,51470.276
1970,7.5883,622,16.6667,89,13,51918.389
1971,7.1934,551,16.7667,112,12,52431.647
1972,6.2049,536,14.9833,158,11,52894.183
1973,6.6367,376,17.0667,123,10,53332.805
1974,6.2941,574,16.3,184,9,53689.61
1975,7.292,572,16.95,171,8,53955.042
1976,7.1211,418,17.65,247,7,54159.049
1977,6.2587,821,15.5833,87,6,54378.362
1978,7.186,763,15.8167,51,5,54602.193
```

The data file can be any type of file that R can read. In this case it is another .csv file. This file needs a synonym in order to be used in a report request. You create the synonym for this file using the Adapter for Delimited Files.

The following is the generated Master File, wine_forecast.mas.

```
FILENAME=WINE_FORECAST, SUFFIX=DFIX    , CODEPAGE=1252,
   DATASET=prediction/wine_forecast.csv, $
SEGMENT=WINE_FORECAST, SEGTYPE=S0, $
    FIELDNAME=YEAR1, ALIAS=Year, USAGE=I6, ACTUAL=A5V,
      MISSING=ON,      TITLE='Year', $
    FIELDNAME=PRICE, ALIAS=Price, USAGE=D8.4, ACTUAL=A7V,
      MISSING=ON,      TITLE='Price', $
    FIELDNAME=WINTERRAIN, ALIAS=WinterRain, USAGE=I5, ACTUAL=A3V,
      MISSING=ON,      TITLE='WinterRain', $
    FIELDNAME=AGST, ALIAS=AGST, USAGE=D9.4, ACTUAL=A8V,
      MISSING=ON,      TITLE='AGST', $
    FIELDNAME=HARVESTRAIN, ALIAS=HarvestRain, USAGE=I5, ACTUAL=A3V,
      MISSING=ON,      TITLE='HarvestRain', $
    FIELDNAME=AGE, ALIAS=Age, USAGE=I4, ACTUAL=A2V,      MISSING=ON,
TITLE='Age', $
    FIELDNAME=FRANCEPOP, ALIAS=FrancePop, USAGE=D11.3, ACTUAL=A11V,
      MISSING=ON,      TITLE='FrancePop', $
```

The following is the generated Access File, wine_forecast.acx.

```
SEGNAME=WINE_FORECAST,   DELIMITER=',',   ENCLOSURE=",   HEADER=YES,
CDN=COMMAS_DOT,   CONNECTION=<local>, $
```

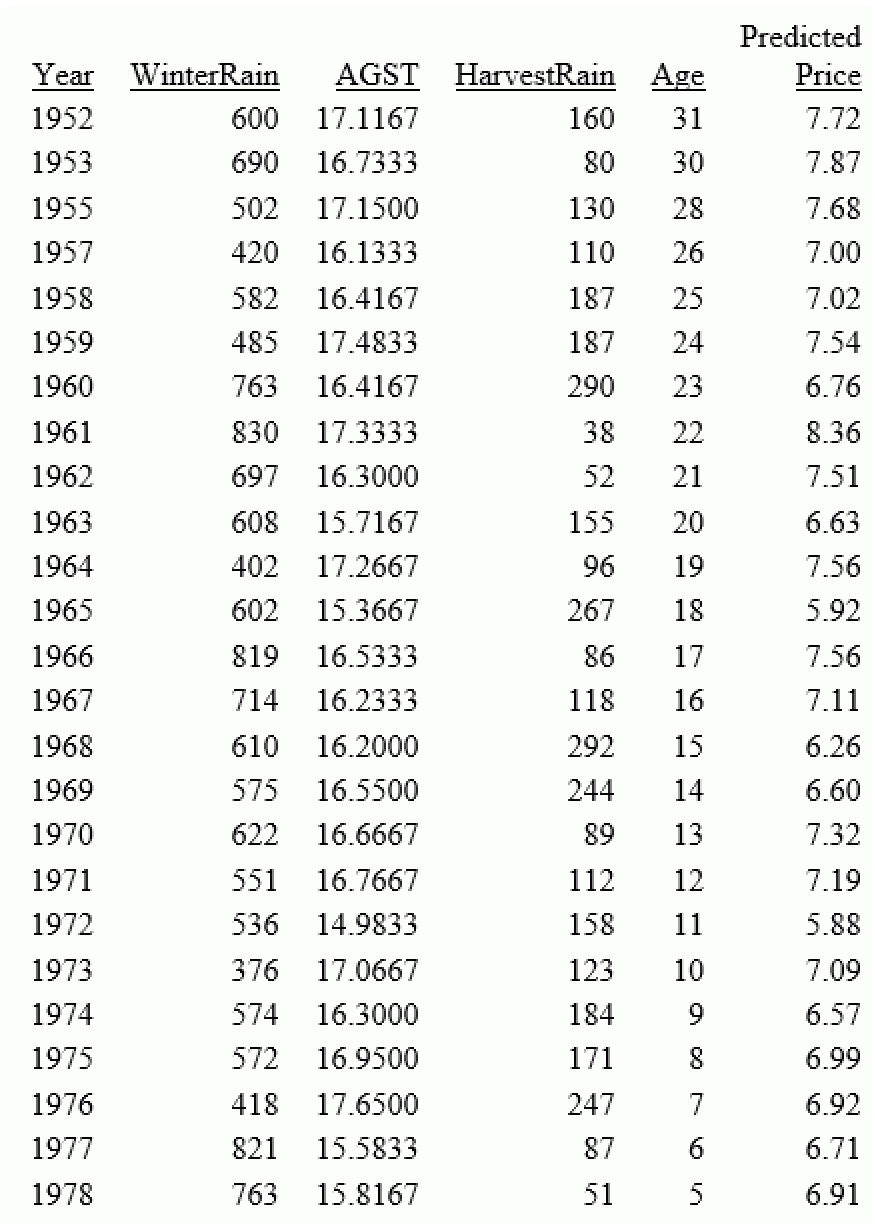
The following request, wine_forecast_price_report.fex, uses the RSERVE bulit-in function to run the script and return a report.

```
-*wine_forecast_price_report.fex
TABLE FILE PREDICTION/WINE_FORECAST
PRINT
  YEAR
  WINTERRAIN
  AGST
  HARVESTRAIN
  AGE

  COMPUTE PREDICTED_PRICE/D18.2 MISSING ON ALL=
    RSERVE(prediction/wine_run_model, AGST, HARVESTRAIN, WINTERRAIN, AGE, Price);  AS
'Predicted,Price'

ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image.



Year	WinterRain	AGST	HarvestRain	Age	Predicted Price
1952	600	17.1167	160	31	7.72
1953	690	16.7333	80	30	7.87
1955	502	17.1500	130	28	7.68
1957	420	16.1333	110	26	7.00
1958	582	16.4167	187	25	7.02
1959	485	17.4833	187	24	7.54
1960	763	16.4167	290	23	6.76
1961	830	17.3333	38	22	8.36
1962	697	16.3000	52	21	7.51
1963	608	15.7167	155	20	6.63
1964	402	17.2667	96	19	7.56
1965	602	15.3667	267	18	5.92
1966	819	16.5333	86	17	7.56
1967	714	16.2333	118	16	7.11
1968	610	16.2000	292	15	6.26
1969	575	16.5500	244	14	6.60
1970	622	16.6667	89	13	7.32
1971	551	16.7667	112	12	7.19
1972	536	14.9833	158	11	5.88
1973	376	17.0667	123	10	7.09
1974	574	16.3000	184	9	6.57
1975	572	16.9500	171	8	6.99
1976	418	17.6500	247	7	6.92
1977	821	15.5833	87	6	6.71
1978	763	15.8167	51	5	6.91

STDDEV: Calculating the Standard Deviation for a Set of Data Values

The STDDEV function returns a numeric value that represents the amount of dispersion in the data. The set of data can be specified as the entire population or a sample. The standard deviation is the square root of the variance, which is a measure of how observations deviate from their expected value (mean). If specified as a population, the divisor in the standard deviation calculation (also called degrees of freedom) will be the total number of data points, N. If specified as a sample, the divisor will be N-1.

If x_i is an observation, N is the number of observations, and μ is the mean of all of the observations, the formula for calculating the standard deviation for a population is:

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_1-\mu)^2}$$

To calculate the standard deviation for a sample, the mean is calculated using the sample observations, and the divisor is N-1 instead of N.

Reference: Calculate the Standard Deviation in a Set of Data

```
STDDEV(field, sampling)
```

where:

field

Numeric

Is the set of observations for the standard deviation calculation.

sampling

Keyword

Indicates the origin of the data set. Can be one of the following values.

- **P** Entire population.
- **S** Sample of population.

Note: Arguments for STDDEV cannot be prefixed fields. If you need to work with fields that have a prefix operator applied, apply the prefix operators to the fields in COMPUTE commands and save the results in a HOLD file. Then, run the standard deviation against the HOLD file.

Example: **Calculating a Standard Deviation**

STDDEV calculates the standard deviation of DOLLARS.

```
STDDEV(DOLLARS,S)
```

The result is 6,157.711080272.

Chapter 16

Simplified System Functions

Simplified system functions have streamlined parameter lists, similar to those used by SQL functions. In some cases, these simplified functions provide slightly different functionality than previous versions of similar functions.

The simplified functions do not have an output argument. Each function returns a value that has a specific data type.

When used in a request against a relational data source, these functions are optimized (passed to the RDBMS for processing).

In this chapter:

EDAPRINT: Inserting a Custom Message in the EDAPRINT Log File

Syntax: How to Insert a Message in the EDAPRINT Log File

```
EDAPRINT(message_type, 'message')
```

where:

message_type

Keyword

Can be one of the following message types.

- **I.** Informational message.
- **W.** Warning message.
- **E.** Error message.

'message'

Is the message to insert, enclosed in single quotation marks.

Example: **Inserting a Custom Message in the EDAPRINT Log File**

The following procedure inserts three messages in the EDAPRINT log file.

```
-SET &I = EDAPRINT(I, 'This is a test informational message');
-SET &W = EDAPRINT(W, 'This is a test warning message');
-SET &E = EDAPRINT(E, 'This is a test error message');
```

ENCRYPT: Encrypting a Password

The ENCRYPT function encrypts an alphanumeric input value using the encryption algorithm configured in the server. The result is returned as variable length alphanumeric.

Syntax: **How to Encrypt a Password**

```
ENCRYPT(password)
```

where:

password

Fixed length alphanumeric

Is the value to be encrypted.

Example: **Encrypting a Password**

ENCRYPT encrypts the password *guestpassword*.

```
ENCRYPT('guestpassword')
```

The returned encrypted value is {AES}963AFA754E1763ABE697E8C5E764115E.

GETENV: Retrieving the Value of an Environment Variable

The GETENV function takes the name of an environment variable and returns its value as a variable length alphanumeric value.

Syntax: **How to Retrieve the Value of an Environment Variable**

```
GETENV(var_name)
```

where:

var_name
: fixed length alphanumeric

Is the name of the environment variable whose value is being retrieved.

Example: Retrieving the Value of an Environment Variable

GETENV retrieves the value of the server variable EDAEXTSEC.

```
GETENV('EDAEXTSEC')
```

The value returned is ON if the server was started with security on or OFF if the server was started with security off.

PUTENV: Assigning a Value to an Environment Variable

The PUTENV function assigns a value to an environment variable. The function returns an integer return code whose value is 1 (one) if the assignment is not successful or 0 (zero) if it is successful.

Syntax: How to Assign a Value to an Environment Variable

```
PUTENV(var_name, var_value)
```

where:

var_name
: Fixed length alphanumeric

Is the name of the environment variable to be set.

var_value
: Alphanumeric

Is the value you want to assign to the variable.

Example: Assigning a Value to the UNIX PS1 Variable

PUTENV assigns the value *FOCUS/Shell:* to the UNIX PS1 variable.

```
PUTENV('PS1','FOCUS/Shell:')
```

This causes UNIX to display the following prompt when the user issues the UNIX shell command SH:

```
FOCUS/Shell:
```

SLACK: Posting a Message to a Slack Channel

SLACK posts a message to a Slack channel from a WebFOCUS procedure:

- If the message is sent successfully, the function returns the value *true*.
- If the message is not sent successfully, the function returns a blank.

Syntax: **How to Post a Message to a Slack Channel**

```
SLACK(workspace, channel, message)
```

where:

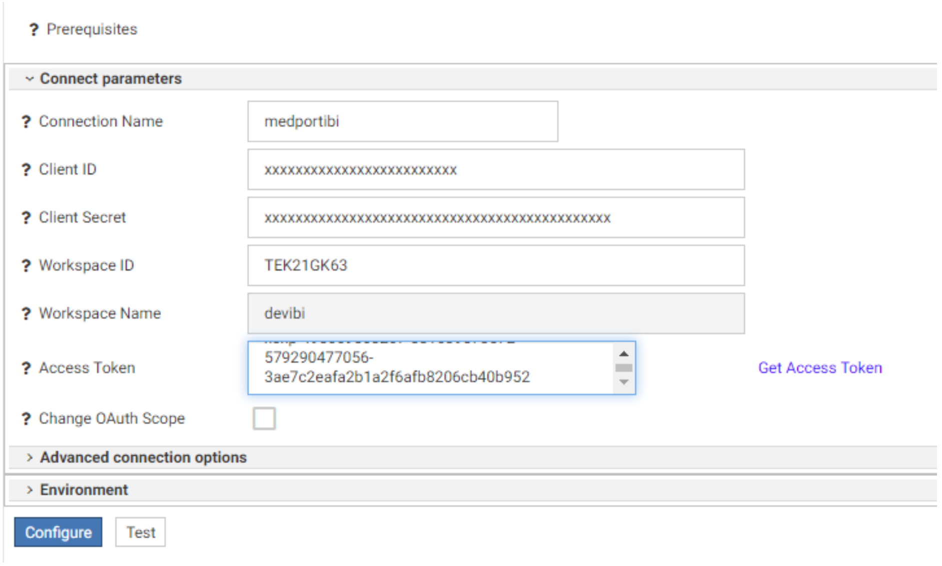
workspace
Is a Workspace name.

channel
Is a Channel name.

message
Is an alphanumeric field containing the message.

Example: **Sending a Slack Message From a WebFOCUS Request**

The Adapter for Slack has been configured to have a connection to the devibi workspace, as shown in the following image.

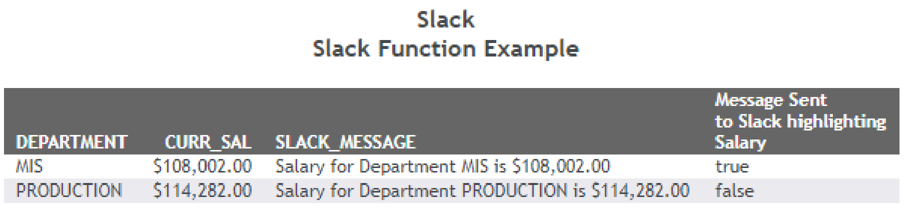


Connection Name	medportibi
Client ID	xxxxxxxxxxxxxxxxxxxxxxxxx
Client Secret	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Workspace ID	TEK21GK63
Workspace Name	devibi
Access Token	579290477056-3ae7c2eafa2b1a2f6afb8206cb40b952
Change OAuth Scope	☐

The following request sends a Slack message to the *general* channel of the *devibi* Workspace, when the department is MIS.

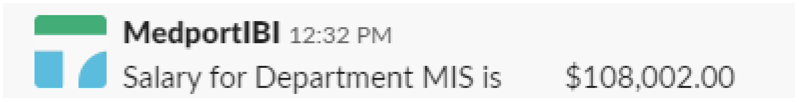
```
TABLE FILE ibisamp/EMPLOYEE
SUM
  CURR_SAL
  AND COMPUTE SLACK_MESSAGE/A200 = 'Salary for Department ' | DEPARTMENT ||
' is ' | LJUST(20, FPRINT(CURR_SAL,'D12.2M'), 'A20');
  AND COMPUTE CURR_SAL_SLACK/A20=IF DEPARTMENT EQ 'MIS'
      THEN SLACK('devibi', 'general', SLACK_MESSAGE) ELSE 'false';
      AS 'Message Sent,to Slack highlighting,Salary'
BY DEPARTMENT
HEADING
"Slack"
"Slack Function Example"
ON TABLE SET PAGE-NUM NOLEAD
ON TABLE NOTOTAL
ON TABLE SET STYLE *
INCLUDE=IBFS:/FILE/IBI_HTML_DIR/javaassist/intl/EN/ENIADefault_combine.sty,
$
ENDSTYLE
END
```

The output is shown in the following image.


Slack
Slack Function Example

DEPARTMENT	CURR_SAL	SLACK_MESSAGE	Message Sent to Slack highlighting Salary
MIS	$108,002.00	Salary for Department MIS is $108,002.00	true
PRODUCTION	$114,282.00	Salary for Department PRODUCTION is $114,282.00	false

The message in the Slack channel is shown in the following image.


MedportIBI 12:32 PM
Salary for Department MIS is $108,002.00

Chapter 17

System Functions

System functions call the operating system to obtain information about the operating environment or to use a system service.

In this chapter:

CLSDDREC: Closing All Files Opened by the PUTDDREC Function

The CLSDDREC function closes all files opened by the PUTDDREC function. If PUTDDREC is called in a Dialogue Manager -SET command, the files opened by PUTDDREC are not closed automatically until the end of a request or connection. In this case, you can close the files and free the memory used to store information about open file by calling the CLSDDREC function.

Syntax: How to Close All Files Opened by the PUTDDREC Function

```
CLSDDREC(output)
```

where:

output

Integer

Is the return code, which can be one of the following values:

- **0,** which indicates that the files are closed.
- **1,** which indicates an error while closing the files.

Example: Closing Files Opened by the PUTDDREC Function

This example closes files opened by the PUTDDREC function:

```
CLSDDREC('I1')
```

FEXERR: Retrieving an Error Message

The FEXERR function retrieves an Information Builders error message. It is especially useful in a procedure using a command that suppresses the display of output messages.

An error message consists of up to four lines of text. The first line contains the message and the remaining three contain a detailed explanation, if one exists. FEXERR retrieves the first line of the error message.

Syntax: How to Retrieve an Error Message

```
FEXERR(error, 'A72')
```

where:

`error`

Numeric

Is the error number, up to 5 digits long.

`'A72'`

Is the format of the output value. The format is A72, the maximum length of an Information Builders error message.

Example: Retrieving an Error Message

FEXERR retrieves the error message whose number is contained in the &ERR variable, in this case 650. The result has the format A72.

```
FEXERR(&ERR, 'A72')
```

The result is (FOC650) THE DISK IS NOT ACCESSED.

FGETENV: Retrieving the Value of an Environment Variable

The FGETENV function retrieves the value of an environment variable and returns it as an alphanumeric string.

Syntax: How to Retrieve the Value of an Environment Variable

```
FGETENV(length, 'varname', outlen, output)
```

where:

length

Integer

Is the number of characters in the environment variable name.

varname

Alphanumeric

Is the name of the environment variable whose value is being retrieved.

outlen

Integer

Is the length of the environment variable value returned.

output

Alphanumeric

Is the format of the field in which the environment variable's value is stored.

FPUTENV: Assigning a Value to an Environment Variable

Available Operating Systems: IBM i (formerly referred to as i5/OS), Tandem, UNIX, Windows

The FPUTENV function assigns a character string to an environment variable.

Limit: You cannot use FPUTENV to set or change FOCPRINT, FOCPATH, or USERPATH. Once started, these variables are held in memory and not reread from the environment.

Syntax: How to Assign a Value to an Environment Variable

```
FPUTENV (varname_length,'varname',value_length, 'value', output)
```

where:

varname_length

Integer

Is the maximum number of characters in the name of the environment variable.

varname

Alphanumeric

Is the name of the environment variable. The name must be right-justified and padded with blanks to the maximum length specified by *varname_length*.

value_length

Is the maximum length of the environment variable value.

Note: The sum of *varname_length* and *value_length* cannot exceed 64.

value

Alphanumeric

Is the value you wish to assign to the environment variable. The string must be right-justified and contain no embedded blanks. Strings that contain embedded blanks are truncated at the first blank.

output

Integer

Is the return code. If the variable is set successfully, the return code is 0. Any other value indicates a failure occurred.

Example: Assigning a Value to an Environment Variable

FPUTENV assigns the value FOCUS/Shell to the PS1 variable and stores it in a field with the format A12:

```
-SET &RC = FPUTENV(3,'PS1', 12, 'FOCUS/Shell:', 'I4');
```

The request displays the following prompt when the user issues the UNIX shell command SH:

```
FOCUS/Shell:
```

GETUSER: Retrieving a User ID

The GETUSER function retrieves the ID of the connected user.

Syntax: **How to Retrieve a User ID**

```
GETUSER(output)
```

where:

output

Alphanumeric, at least A8

Is the result field, whose length depends on the platform on which the function is issued. Provide a length as long as required for your platform; otherwise the output will be truncated.

Example: **Retrieving a User ID**

GETUSER retrieves the user ID of the person running the flow.

```
GETUSER(USERID)
```

JOBNAME: Retrieving the Current Process Identification String

The JOBNAME function retrieves the raw identification string of the current process from the operating system. This is also commonly known as a process PID at the operating system level. The function is valid in all environments, but is typically used in Dialogue Manager and returns the value as an alphanumeric string (even though a PID is pure numeric on some operating systems).

Note: JOBNAME strings differ between some operating systems in terms of look and length. For example, Windows, UNIX, and z/OS job names are pure numeric (typically a maximum of 8 characters long), while an IBM i job name is a three-part string that has a 26 character maximum length. Since an application may eventually be run in another (unexpected) environment in the future, it is good practice to use the maximum length of 26 to avoid accidental length truncation in the future. Applications using this function for anything more than simple identification may also need to account for the difference in the application code.

Syntax: How to Retrieve the Current Process Identification String

```
JOBNAME(length, output)
```

where:

length

Integer

Is the maximum number of characters to return from the PID system call.

output

Alphanumeric

Is the returned process identification string, whose length depends on the platform on which the function is issued. Provide a length as long as required for your platform. Otherwise, the output will be truncated.

Example: Retrieving a Process Identification String

The following example uses the JOBNAME function to retrieve the current process identification string to an A26 string and then truncate it for use in a -TYPE statement:

```
-SET &JOBNAME = JOBNAME(26, 'A26');
-SET &JOBNAME = TRUNCATE(&JOBNAME);
-TYPE The Current system PID &JOBNAME is processing.
```

For example, on Windows, the output is similar to the following:

```
The Current system PID 2536 is processing.
```

PUTDDREC: Writing a Character String as a Record in a Sequential File

The PUTDDREC function writes a character string as a record in a sequential file. The file must be identified with a FILEDEF (DYNAM on z/OS) command. If the file is defined as an existing file (with the APPEND option), the new record is appended. If the file is defined as NEW and it already exists, the new record overwrites the existing file.

PUTDDREC opens the file if it is not already open. Each call to PUTDDREC can use the same file or a new one. All of the files opened by PUTDDREC remain open until the end of a request or connection. At the end of the request or connection, all files opened by PUTDDREC are automatically closed.

If PUTDDREC is called in a Dialogue Manager -SET command, the files opened by PUTDDREC are not closed automatically until the end of a request or connection. In this case, you can close the files and free the memory used to store information about open file by calling the CLSDDREC function.

Syntax: How to Write a Character String as a Record in a Sequential File

```
PUTDDREC(ddname, dd_len, record_string, record_len, output)
```

where:

ddname

Alphanumeric

Is the logical name assigned to the sequential file in a FILEDEF command.

dd_len

Numeric

Is the number of characters in the logical name.

record_string

Alphanumeric

Is the character string to be added as the new record in the sequential file.

record_len

Numeric

Is the number of characters to add as the new record.

It cannot be larger than the number of characters in *record_string*. To write all of *record_string* to the file, *record_len* should equal the number of characters in *record_string* and should not exceed the record length declared in the command. If *record_len* is shorter than the declared length declared, the resulting file may contain extraneous characters at the end of each record. If *record_string* is longer than the declared length, *record_string* may be truncated in the resulting file.

`output`

Integer

Is the return code, which can have one of the following values:

`0` - Record is added.
`-1` - FILEDEF statement is not found.
`-2` - Error while opening the file.
`-3` - Error while adding the record to the file.

Example: Writing a Character String as a Record in a Sequential File

Using the CAR synonym as input,

```
FILEDEF LOGGING DISK baseapp/logging.dat
```

```
PUTDDREC('LOGGING', 7, 'Country:' | COUNTRY, 20, 'I5')
```

would return the value 0, and would write the following lines to logging.dat:

Country: ENGLAND

Country: JAPAN

Country: ITALY

Country: W GERMANY

Country: FRANCE

SLEEP: Suspending Execution for a Given Number of Seconds

The SLEEP function suspends execution for the number of seconds you specify as its input argument.

This function is only supported in Dialogue Manager. It is useful when you need to wait to start a specific procedure or application.

Syntax: How to Suspend Execution for a Specified Number of Seconds

```
SLEEP(delay, output);
```

where:

delay

Numeric

Is the number of seconds to delay execution. The number can be specified down to the millisecond.

output

Numeric

The value returned is the same value you specify for delay.

Example: Suspending Execution for Four Seconds

SLEEP suspends execution for four seconds:

```
-SET &DELAY = SLEEP(4.0, 'I2');
```

SYSVAR: Retrieving the Value of a z/OS System Variable

Available Operating Systems: z/OS

The SYSVAR function populates a Dialogue Manager amper variable with the contents of any z/OS system variable. System variables are in the format [&]*name*[.], where the dot is optional. They can be provided by the operating system or can be user defined. The function can be called in a -SET command.

Syntax: How to Retrieve the Value of a z/OS System Variable

```
-SET &dmvar = SYSVAR('length','[&]sysvar[.]','outfmt');
```

where:

&dmvar

Alphanumeric

Is the name of the Dialogue Manager variable to be populated with the value of the z/OS system variable.

`length`

Alphanumeric

Is the length of the next parameter in the call. Do not include the escape character in the length, if one is present in the *sysvar* argument.

`[&|]sysvar[.]`

Alphanumeric

Is the name of the system variable to be retrieved. Note that the ampersand (&) and the dot (.) are optional. If the ampersand is included, it must be followed by the escape character (|).

`outfmt`

Alphanumeric

Is the format of the returned value enclosed in single quotation marks.

Example: Retrieving the Value of the z/OS SYSNAME Variable

The following example populates the Dialogue Manager variable named &MYSNAME2 with the value of the z/OS SYSNAME variable:

```
-SET &MYSNAME2=SYSVAR('7','SYSNAME','A8');
-TYPE SYSNAME:&MYSNAME2
```

The output is similar to the following:

```
SYSNAME:IBI1
```

Chapter 18

Simplified Geography Functions

The simplified geography functions perform location-based calculations and retrieve geocoded points for various types of location data. They are used by the WebFOCUS location intelligence products that produce maps and charts. Some of the geography functions use GIS services and require valid credentials for accessing Esri ArcGIS proprietary data.

In this chapter:

Sample Geography Files

Some of the examples for the geography functions use geography sample files. One file, esri-citibke.csv has station names, latitudes and longitudes, and trip start times and end times. The other file, esri-geo10036.ftm has geometry data. To run the examples that use these files, create an application named *esri*, and place the following files into the application folder.

esri-citibike.mas

```
FILENAME=ESRI-CITIBIKE, SUFFIX=DFIX    ,
 DATASET=esri/esri-citibike.csv, $
  SEGMENT=CITIBIKE_TRIPDATA, SEGTYPE=S0, $
    FIELDNAME=TRIPDURATION, ALIAS=tripduration, USAGE=I7, ACTUAL=A5V,
      TITLE='tripduration', $
    FIELDNAME=STARTTIME, ALIAS=starttime, USAGE=HMDYYS, ACTUAL=A18,
      TITLE='starttime', $
    FIELDNAME=STOPTIME, ALIAS=stoptime, USAGE=HMDYYS, ACTUAL=A18,
      TITLE='stoptime', $
    FIELDNAME=START_STATION_ID, ALIAS='start station id', USAGE=I6, ACTUAL=A4V,
      TITLE='start station id', $
    FIELDNAME=START_STATION_NAME, ALIAS='start station name', USAGE=A79V,
      ACTUAL=A79BV, TITLE='start station name', $
    FIELDNAME=START_STATION_LATITUDE, ALIAS='start station latitude', USAGE=P20.15,
      ACTUAL=A18V, TITLE='start station latitude',
      GEOGRAPHIC_ROLE=LATITUDE,  $
    FIELDNAME=START_STATION_LONGITUDE, ALIAS='start station longitude', USAGE=P20.14,
      ACTUAL=A18V, TITLE='start station longitude',
      GEOGRAPHIC_ROLE=LONGITUDE,  $
    FIELDNAME=END_STATION_ID, ALIAS='end station id', USAGE=I6,
      ACTUAL=A4V, TITLE='end station id', $

    FIELDNAME=END_STATION_NAME, ALIAS='end station name', USAGE=A79V,
      ACTUAL=A79BV, TITLE='end station name', $
    FIELDNAME=END_STATION_LATITUDE, ALIAS='end station latitude', USAGE=P20.15,
      ACTUAL=A18V, TITLE='end station latitude',
      GEOGRAPHIC_ROLE=LATITUDE,  $
    FIELDNAME=END_STATION_LONGITUDE, ALIAS='end station longitude', USAGE=P20.14,
      ACTUAL=A18V, TITLE='end station longitude',
      GEOGRAPHIC_ROLE=LONGITUDE,  $
    FIELDNAME=BIKEID, ALIAS=bikeid, USAGE=I7, ACTUAL=A5,
      TITLE='bikeid', $
    FIELDNAME=USERTYPE, ALIAS=usertype, USAGE=A10V, ACTUAL=A10BV,
      TITLE='usertype', $
    FIELDNAME=BIRTH_YEAR, ALIAS='birth year', USAGE=I6, ACTUAL=A4,
      TITLE='birth year', $
    FIELDNAME=GENDER, ALIAS=gender, USAGE=I3, ACTUAL=A1,
      TITLE='gender', $
 SEGMENT=ESRIGEO, SEGTYPE=KU, SEGSUF=FIX, PARENT=CITIBIKE_TRIPDATA,
    DATASET=esri/esri-geo10036.ftm (LRECL 80 RECFM V, CRFILE=ESRI-GEO10036, $
```

esri-citibike.acx

```
SEGNAME=CITIBIKE_TRIPDATA,
  DELIMITER=',',
  ENCLOSURE=",
  HEADER=NO,
  CDN=OFF, $
```

esri-citibike.csv

Note: Each complete record must be on a single line. Therefore, you must remove any line breaks that may have been inserted due to the page width in this document.

```
1094,11/1/2015 0:00,11/1/2015 0:18,537,Lexington Ave & E 24 St,
40.74025878,-73.98409214,531,Forsyth St & Broome St,
40.71893904,-73.99266288,23959,Subscriber,1980,1

520,11/1/2015 0:00,11/1/2015 0:08,536,1 Ave & E 30 St,
40.74144387,-73.97536082,498,Broadway & W 32 St,
40.74854862,-73.98808416,22251,Subscriber,1988,1

753,11/1/2015 0:00,11/1/2015 0:12,229,Great Jones St,
40.72743423,-73.99379025,328,Watts St & Greenwich St,
40.72405549,-74.00965965,15869,Subscriber,1981,1

353,11/1/2015 0:00,11/1/2015 0:06,285,Broadway & E 14 St,
40.73454567,-73.99074142,151,Cleveland Pl & Spring St,
40.72210379,-73.99724901,21645,Subscriber,1987,1

1285,11/1/2015 0:00,11/1/2015 0:21,268,Howard St & Centre St,
40.71910537,-73.99973337,476,E 31 St & 3 Ave,40.74394314,-73.97966069,14788,Customer,,0

477,11/1/2015 0:00,11/1/2015 0:08,379,W 31 St & 7 Ave,40.749156,-73.9916,546,E 30 St &
Park Ave S,40.74444921,-73.98303529,21128,Subscriber,1962,2

362,11/1/2015 0:00,11/1/2015 0:06,407,Henry St & Poplar St,
40.700469,-73.991454,310,State St & Smith St,40.68926942,-73.98912867,21016,Subscriber,
1978,1

2316,11/1/2015 0:00,11/1/2015 0:39,147,Greenwich St & Warren St,
40.71542197,-74.01121978,441,E 52 St & 2 Ave,40.756014,-73.967416,24117,Subscriber,
1988,2

627,11/1/2015 0:00,11/1/2015 0:11,521,8 Ave & W 31 St,
40.75096735,-73.99444208,285,Broadway & E 14 St,
40.73454567,-73.99074142,17048,Subscriber,1986,2

1484,11/1/2015 0:01,11/1/2015 0:26,281,Grand Army Plaza & Central Park S,
40.7643971,-73.97371465,367,E 53 St & Lexington Ave,
40.75828065,-73.97069431,16779,Customer,,0
```

```
284,11/1/2015 0:01,11/1/2015 0:06,247,Perry St & Bleecker St,
40.73535398,-74.00483091,453,W 22 St & 8 Ave,40.74475148,-73.99915362,17272,Subscriber,
1976,1

886,11/1/2015 0:01,11/1/2015 0:16,492,W 33 St & 7 Ave,40.75019995,-73.99093085,377,6
Ave & Canal St,40.72243797,-74.00566443,23019,Subscriber,1982,1

1379,11/1/2015 0:01,11/1/2015 0:24,512,W 29 St & 9 Ave,40.7500727,-73.99839279,445,E
10 St & Avenue A,40.72740794,-73.98142006,23843,Subscriber,1962,2

179,11/1/2015 0:01,11/1/2015 0:04,319,Fulton St & Broadway,
40.711066,-74.009447,264,Maiden Ln & Pearl St,
40.70706456,-74.00731853,22538,Subscriber,1981,1

309,11/1/2015 0:01,11/1/2015 0:07,160,E 37 St & Lexington Ave,
40.748238,-73.978311,362,Broadway & W 37 St,40.75172632,-73.98753523,22042,Subscriber,
1988,1

616,11/1/2015 0:02,11/1/2015 0:12,479,9 Ave & W 45 St,40.76019252,-73.9912551,440,E 45
St & 3 Ave,40.75255434,-73.97282625,22699,Subscriber,1982,1

852,11/1/2015 0:02,11/1/2015 0:16,346,Bank St & Hudson St,
40.73652889,-74.00618026,375,Mercer St & Bleecker St,
40.72679454,-73.99695094,21011,Subscriber,1991,1

1854,11/1/2015 0:02,11/1/2015 0:33,409,DeKalb Ave & Skillman St,
40.6906495,-73.95643107,3103,N 11 St & Wythe Ave,
40.72153267,-73.95782357,22011,Subscriber,1992,1

1161,11/1/2015 0:02,11/1/2015 0:21,521,8 Ave & W 31 St,40.75096735,-73.99444208,461,E
20 St & 2 Ave,40.73587678,-73.98205027,19856,Subscriber,1957,1

917,11/1/2015 0:02,11/1/2015 0:17,532,S 5 Pl & S 4 St,40.710451,-73.960876,393,E 5 St
& Avenue C,40.72299208,-73.97995466,18598,Subscriber,1991,1
```

esri-geo10036.mas

```
FILENAME=ESRI-GEO10036, SUFFIX=FIX     ,
 DATASET=esri/esri-geo10036.ftm (LRECL 80 RECFM V, IOTYPE=STREAM, $
  SEGMENT=ESRIGEO, SEGTYPE=S0, $
    FIELDNAME=GEOMETRY, ALIAS=GEOMETRY, USAGE=TX80L, ACTUAL=TX80,
      MISSING=ON, $
```

esri-geo10036.ftm

```
{"rings":[[[-73.9803889998524,40.7541490002762],[-73.9808779999197,40.7534830001
404],[-73.9814419998484,40.7537140000011],[-73.9824040001445,40.7541199998382],[
-73.982461000075,40.7541434001978],[-73.9825620002361,40.7541850001377],[-73.983
2877000673,40.7544888999428],[-73.9833499997027,40.7545150000673],[-73.983644399
969,40.7546397998869],[-73.9836849998628,40.7546570003204],[-73.9841276003085,40
.7548161002829],[-73.984399700086,40.7544544999752],[-73.9846140004357,40.754165
0001147],[-73.984871999743,40.7542749997914],[-73.9866590003126,40.7550369998577
],[-73.9874449996869,40.7553720000178],[-73.9902640001834,40.756570999552],[-73.
9914340001789,40.7570449998269],[-73.9918260002697,40.7572149995726],[-73.992429
0001982,40.7574769999636],[-73.9927679996434,40.7576240004473],[-73.993069000034
3,40.7578009996165],[-73.9931059999419,40.7577600004237],[-73.9932120003335,40.7
576230004012],[-73.9933250001486,40.7576770001934],[-73.9935390001247,40.7577669

998472],[-73.993725999755,40.7578459998931],[-73.9939599997542,40.757937999639],
[-73.9940989998689,40.7579839999617],[-73.9941529996611,40.7579959996157],[-73.9
942220001452,40.7580159996387],[-73.9943040003293,40.7580300002843],[-73.9943650
004444,40.7580330004227],[-73.99446499966,40.7580369997078],[-73.9945560002591,4
0.7580300002843],[-73.9946130001898,40.7580209998693],[-73.9945689999594,40.7580
809999383],[-73.9945449997519,40.7581149997075],[-73.9944196999092,40.7582882001
404],[-73.9943810002829,40.7583400001909],[-73.9953849998179,40.7587409997973],[
-73.9959560000693,40.7589690004191],[-73.9960649996999,40.7590149998424],[-73.99
68730000888,40.7593419996336],[-73.996975000296,40.7593809996335],[-73.997314999
7874,40.7595379996789],[-73.9977009996014,40.7597030000935],[-73.998039999946,40
.7598479995856],[-73.998334000014,40.7599709998618],[-73.9987769997587,40.760157
0003453],[-73.9990089996656,40.7602540003219],[-74.0015059997021,40.761292999672

2],[-74.0016340002089,40.7613299995799],[-74.0015350001401,40.7614539999022],[-7
4.0014580001865,40.7615479997405],[-74.0013640003483,40.7616560002242],[-74.0013
050003255,40.7617199995784],[-74.0011890003721,40.7618369995779],[-74.0010579997
269,40.7619609999003],[-74.0009659999808,40.7620389999],[-74.0008649998198,40.76
21230001764],[-74.0008390004195,40.7621430001993],[-74.0006839995669,40.76226100
0245],[-74.000531999752,40.7623750001062],[-74.0003759997525,40.7624849997829],[
-74.0002840000066,40.7625510001286],[-73.9998659996161,40.762850999574],[-73.999
8279996624,40.7628779999198],[-73.9995749996864,40.7630590001727],[-73.999312000
1487,40.7632720001028],[-73.9991639996189,40.7633989996642],[-73.998941000127,40
.7636250001936],[-73.9987589998279,40.7638580001466],[-73.9986331999622,40.76402
77004181],[-73.9986084002574,40.7640632002565],[-73.9984819996445,40.76423400039
89],[-73.9983469997142,40.7644199999831],[-73.998171999738,40.7646669996823],[-7
3.9980319995771,40.7648580003964],[-73.9979881998955,40.7649204996813],[-73.9979
368000432,40.7649942000224],[-73.9978947999051,40.7650573998791],[-73.9977017001
733,40.7653310995507],[-73.9975810003629,40.765481000348],[-73.9975069996483,40.
7654519999099],[-73.9956019999323,40.7646519998899],[-73.9955379996789,40.764625
0004434],[-73.9954779996099,40.7646030003282],[-73.9949389999348,40.764369000329
1],[-73.9936289997785,40.7638200001929],[-73.9934620001711,40.7637539998473],[-7
3.9931520002646,40.7636270002859],[-73.992701000151,40.7634409998023],[-73.99244
19000736,40.7633312995998],[-73.9898629996777,40.7622390001298],[-73.98861200044
34,40.761714000201],[-73.988021000169,40.761460000179],[-73.987028000242,40.7610
439998808],[-73.9867690998141,40.7609346998765],[-73.9848240002274,40.7601130001
149],[-73.9841635003452,40.7598425002312],[-73.9813259998949,40.7586439998208],[
-73.9805479999902,40.7583159999834],[-73.9793569999256,40.757814000216],[-73.978
1150002071,40.7572939996184],[-73.9785670003668,40.7566709996669],[-73.979014000
2958,40.7560309998308],[-73.9794719998329,40.7554120000638],[-73.9799399998311,4
0.7547649999048],[-73.9802380000836,40.7543610001601],[-73.9803889998524,40.7541
490002762]]]}
%$
```

GIS_DISTANCE: Calculating the Distance Between Geometry Points

The GIS_DISTANCE function uses a GIS service to calculate the distance between two geometry points.

Syntax: How to Calculate the Distance Between Geometry Points

```
GIS_DISTANCE(geo_point1,geo_point2)
```

where:

geo_point1,geo_point2

Fixed length alphanumeric, large enough to hold the JSON describing the point (for example, A200).

Are the geometry points for which you want to calculate the distance.

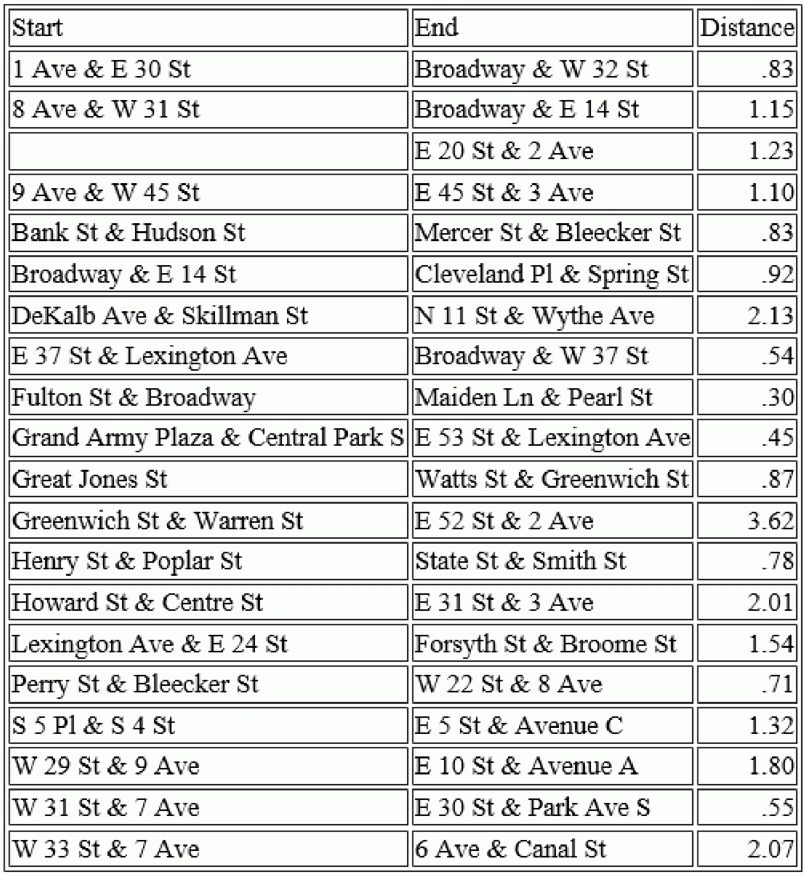
Note: You can generate a geometry point using the GIS_POINT function.

Example: Calculating the Distance Between Two Geometry Points

The following uses a citibike .csv file that contains station names, latitudes and longitudes, and trip start times and end times. It uses the GIS_POINT function to define geometry points for start stations and end stations. It then uses GIS_DISTANCE to calculate the distance between them.

```
DEFINE FILE esri/esri-citibike
STARTPOINT/A200 = GIS_POINT('4326', START_STATION_LONGITUDE,
START_STATION_LATITUDE);
ENDPOINT/A200 = GIS_POINT('4326', END_STATION_LONGITUDE,
END_STATION_LATITUDE);
Distance/P10.2 = GIS_DISTANCE(ENDPOINT, STARTPOINT);
END
TABLE FILE esri/esri-citibike
PRINT END_STATION_NAME AS End Distance
BY START_STATION_NAME AS Start
ON TABLE SET PAGE NOLEAD
END
```

The output is shown in the following image.



Start	End	Distance
1 Ave & E 30 St	Broadway & W 32 St	.83
8 Ave & W 31 St	Broadway & E 14 St	1.15
	E 20 St & 2 Ave	1.23
9 Ave & W 45 St	E 45 St & 3 Ave	1.10
Bank St & Hudson St	Mercer St & Bleecker St	.83
Broadway & E 14 St	Cleveland Pl & Spring St	.92
DeKalb Ave & Skillman St	N 11 St & Wythe Ave	2.13
E 37 St & Lexington Ave	Broadway & W 37 St	.54
Fulton St & Broadway	Maiden Ln & Pearl St	.30
Grand Army Plaza & Central Park S	E 53 St & Lexington Ave	.45
Great Jones St	Watts St & Greenwich St	.87
Greenwich St & Warren St	E 52 St & 2 Ave	3.62
Henry St & Poplar St	State St & Smith St	.78
Howard St & Centre St	E 31 St & 3 Ave	2.01
Lexington Ave & E 24 St	Forsyth St & Broome St	1.54
Perry St & Bleecker St	W 22 St & 8 Ave	.71
S 5 Pl & S 4 St	E 5 St & Avenue C	1.32
W 29 St & 9 Ave	E 10 St & Avenue A	1.80
W 31 St & 7 Ave	E 30 St & Park Ave S	.55
W 33 St & 7 Ave	6 Ave & Canal St	2.07

GIS_DRIVE_ROUTE: Calculating the Driving Directions Between Geometry Points

The GIS_DRIVE_ROUTE function uses a GIS service to calculate the driving route between two geometry points.

Note: This function uses GIS services and requires an Esri ArcGIS adapter connection with named credentials.

Syntax: How to Calculate the Drive Route Between Geometry Points

```
GIS_DRIVE_ROUTE(geo_start_point,geo_end_point)
```

where:

geo_start_point, *geo_point2*

Fixed length alphanumeric, large enough to hold the JSON describing the point (for example, A200).

Is the starting point for which you want to calculate the drive route.

Note: You can generate a geometry point using the GIS_POINT function.

geo_end_point, *geo_point2*

Fixed length alphanumeric, large enough to hold the JSON describing the point (for example, A200).

Is the ending point for which you want to calculate the drive route.

Note: You can generate a geometry point using the GIS_POINT function.

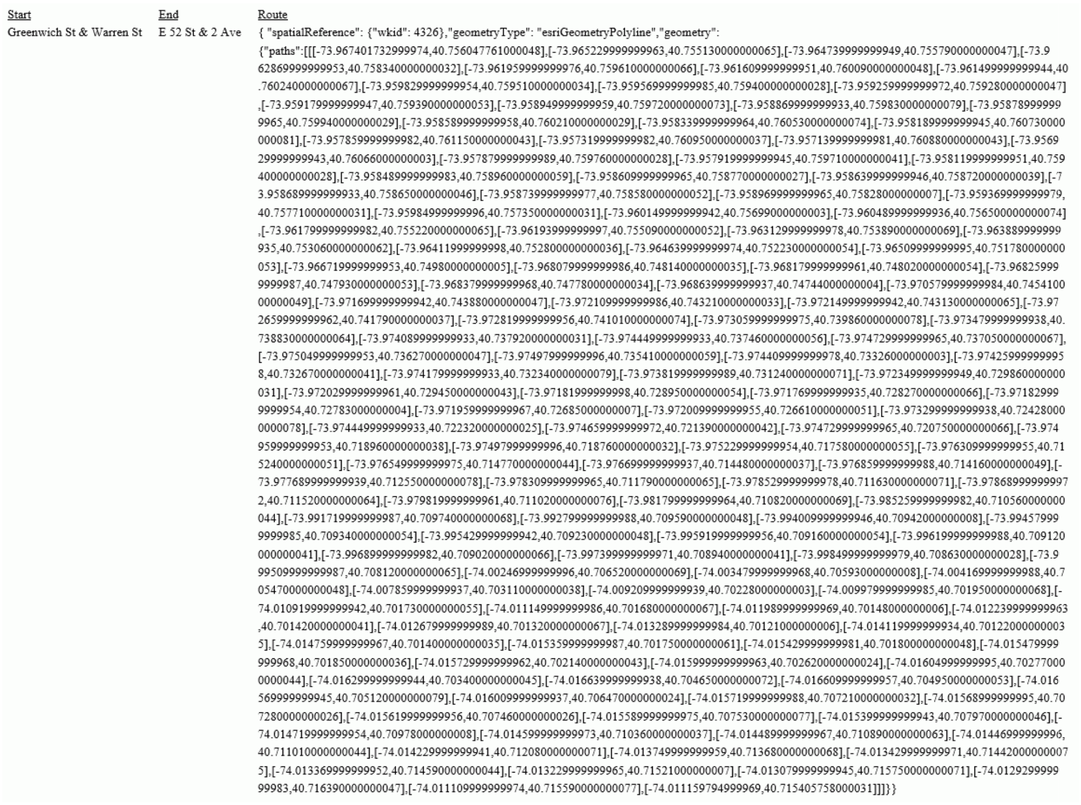
The format of the field to which the drive route will be returned is TX.

Example: Calculating the Drive Route Between Two Geometry Points

The following uses a citibike .csv file that contains station names, latitudes and longitudes, and trip start times and end times. It uses the GIS_POINT function to define geometry points for start stations and end stations. It then uses GIS_DRIVE_ROUTE to calculate the route to get from the end point to the start point.

```
DEFINE FILE esri/esri-citibike
STARTPOINT/A200 = GIS_POINT('4326', START_STATION_LONGITUDE,
START_STATION_LATITUDE);
ENDPOINT/A200 = GIS_POINT('4326', END_STATION_LONGITUDE,
END_STATION_LATITUDE);
Route/TX140 (GEOGRAPHIC_ROLE=GEOMETRY_LINE) =
          GIS_DRIVE_ROUTE(ENDPOINT, STARTPOINT);
END
TABLE FILE esri/esri-citibike
PRINT START_STATION_NAME AS Start END_STATION_NAME AS End Route
WHERE START_STATION_ID EQ 147
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
TYPE=REPORT, GRID=OFF,SIZE-11,$
ENDSTYLE
END
```

The output is shown in the following image.



```
Start                   End              Route
Greenwich St & Warren St  E 52 St & 2 Ave  { "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPolyline","geometry":
                                           {"paths":[[[-73.967401732999974,40.756047761000048],[-73.965229999999963,40.755130000000065],[-73.964739999999949,40.755790000000047],[-73.9
                                           62869999999953,40.758340000000032],[-73.961959999999976,40.759610000000066],[-73.961609999999951,40.760090000000048],[-73.961499999999944,40
                                           .760240000000067],[-73.959829999999954,40.759510000000034],[-73.959569999999985,40.759400000000028],[-73.959259999999972,40.759280000000047]
                                           ,[-73.959179999999947,40.759390000000053],[-73.958949999999959,40.759720000000073],[-73.958869999999933,40.759830000000079],[-73.95878999999
                                           9965,40.759940000000029],[-73.958589999999958,40.760210000000029],[-73.958339999999964,40.760530000000074],[-73.958189999999945,40.760730000
                                           000081],[-73.957859999999982,40.761150000000043],[-73.957319999999982,40.760950000000037],[-73.957139999999981,40.760880000000043],[-73.9569
                                           29999999943,40.76066000000003],[-73.957879999999989,40.759760000000028],[-73.957919999999945,40.759710000000041],[-73.958119999999951,40.759
                                           400000000028],[-73.958489999999983,40.758960000000059],[-73.958609999999965,40.758770000000027],[-73.958639999999946,40.758720000000039],[-7
                                           3.958689999999933,40.758650000000046],[-73.958739999999977,40.758580000000052],[-73.958969999999965,40.75828000000007],[-73.959369999999979,
                                           40.757710000000031],[-73.959849999999996,40.757350000000031],[-73.960149999999942,40.75699000000003],[-73.960489999999936,40.756500000000074]
                                           ,[-73.961799999999982,40.755220000000065],[-73.96193999999997,40.755090000000052],[-73.963129999999978,40.753890000000069],[-73.963889999999
                                           935,40.753060000000062],[-73.96411999999998,40.752800000000036],[-73.964639999999974,40.752230000000054],[-73.96509999999995,40.751780000000
                                           053],[-73.966719999999953,40.74980000000005],[-73.968079999999986,40.748140000000035],[-73.968179999999961,40.748020000000054],[-73.96825999
                                           9999987,40.747930000000053],[-73.968379999999968,40.747780000000034],[-73.968639999999937,40.74744000000004],[-73.970579999999984,40.7454100
                                           00000049],[-73.971699999999942,40.743880000000047],[-73.972109999999986,40.743210000000033],[-73.972149999999942,40.743130000000065],[-73.97
                                           2659999999962,40.741790000000037],[-73.972819999999956,40.741010000000074],[-73.973059999999975,40.739860000000078],[-73.973479999999938,40.
                                           738830000000064],[-73.974089999999933,40.737920000000031],[-73.974449999999933,40.737460000000056],[-73.974729999999965,40.737050000000067],
                                           [-73.975049999999953,40.736270000000047],[-73.97497999999996,40.735410000000059],[-73.974409999999978,40.73326000000003],[-73.97425999999995
                                           8,40.732670000000041],[-73.974179999999933,40.732340000000079],[-73.973819999999989,40.731240000000071],[-73.972349999999949,40.729860000000
                                           031],[-73.972029999999961,40.729450000000043],[-73.97181999999998,40.728950000000054],[-73.971769999999935,40.728270000000066],[-73.97182999
                                           9999954,40.72783000000004],[-73.971959999999967,40.72685000000007],[-73.972009999999955,40.726610000000051],[-73.973299999999938,40.72428000
                                           0000078],[-73.974449999999933,40.722320000000025],[-73.974659999999972,40.721390000000042],[-73.974729999999965,40.720750000000066],[-73.974
                                           959999999953,40.718960000000038],[-73.97497999999996,40.718760000000032],[-73.975229999999954,40.717580000000055],[-73.976309999999955,40.71
                                           5240000000051],[-73.976549999999975,40.714770000000044],[-73.976699999999937,40.714480000000037],[-73.976859999999988,40.714160000000049],[-
                                           73.977689999999939,40.712550000000078],[-73.978309999999965,40.711790000000065],[-73.978529999999978,40.711630000000071],[-73.97868999999997
                                           2,40.711520000000064],[-73.979819999999961,40.711020000000076],[-73.981799999999964,40.710820000000069],[-73.985259999999982,40.710560000000
                                           044],[-73.991719999999987,40.709740000000068],[-73.992799999999988,40.709590000000048],[-73.994009999999946,40.70942000000008],[-73.99457999
                                           9999985,40.709340000000054],[-73.995429999999942,40.709230000000048],[-73.995919999999956,40.709160000000054],[-73.996199999999988,40.709120
                                           000000041],[-73.996899999999982,40.709020000000066],[-73.997399999999971,40.708940000000041],[-73.998499999999979,40.708630000000028],[-73.9
                                           99509999999987,40.708120000000065],[-74.00246999999996,40.706520000000069],[-74.003479999999968,40.70593000000008],[-74.004169999999988,40.7
                                           05470000000048],[-74.007859999999937,40.703110000000038],[-74.009209999999939,40.70228000000003],[-74.009979999999985,40.701950000000068],[-
                                           74.010919999999942,40.701730000000055],[-74.011149999999986,40.701680000000067],[-74.011989999999969,40.70148000000006],[-74.012239999999963
                                           ,40.701420000000041],[-74.012679999999989,40.701320000000067],[-74.013289999999984,40.70121000000006],[-74.014119999999934,40.70122000000003
                                           5],[-74.014759999999967,40.701400000000035],[-74.015359999999987,40.701750000000061],[-74.015429999999981,40.701800000000048],[-74.015479999
                                           999968,40.701850000000036],[-74.015729999999962,40.702140000000043],[-74.015999999999963,40.702620000000024],[-74.01604999999995,40.70277000
                                           0000044],[-74.016299999999944,40.703400000000045],[-74.016639999999938,40.704650000000072],[-74.016609999999957,40.704950000000053],[-74.016
                                           569999999945,40.705120000000079],[-74.016009999999937,40.706470000000024],[-74.015719999999988,40.707210000000032],[-74.01568999999995,40.70
                                           7280000000026],[-74.015619999999956,40.707460000000026],[-74.015589999999975,40.707530000000077],[-74.015399999999943,40.707970000000046],[-
                                           74.014719999999954,40.70978000000008],[-74.014599999999973,40.710360000000037],[-74.014489999999967,40.710890000000063],[-74.01446999999996,
                                           40.711010000000044],[-74.014229999999941,40.712080000000071],[-74.013749999999959,40.713680000000068],[-74.013429999999971,40.71442000000007
                                           5],[-74.013369999999952,40.714590000000044],[-74.013229999999965,40.71521000000007],[-74.013079999999945,40.715750000000071],[-74.0129299999
                                           99983,40.716390000000047],[-74.011109999999974,40.715590000000077],[-74.011159794999969,40.715405758000031]]]}}
```

GIS_GEOCODE_ADDR: Geocoding a Complete Address

GIS_GEOCODE_ADDR uses a GIS geocoding service to obtain the geometry point for a complete address.

Note: This function uses GIS services and requires an Esri ArcGIS adapter connection with named credentials.

Syntax: **How to Geocode a Complete Address**

```
GIS_GEOCODE_ADDR(address[, country])
```

where:

address

Fixed length alphanumeric

Is the complete address to be geocoded.

country

Fixed length alphanumeric

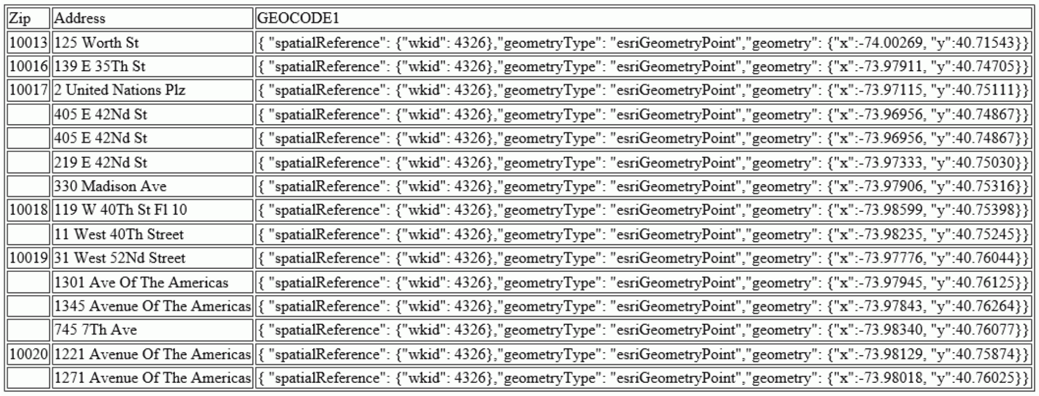
Is a country name, which is optional if the country is the United States.

Example: **Geocoding a Complete Address**

The following request creates a complete address by concatenating the street address, city, state, and ZIP code. It then uses GIS_GEOCODE_ADDR to create a GIS point for the address.

```
DEFINE FILE WF_RETAIL_LITE
GADDRESS/A200 =ADDRESS_LINE_1 || '  ' | CITY_NAME || '  ' | STATE_PROV_NAME
|| ' ' | POSTAL_CODE;
GEOCODE1/A200 = GIS_GEOCODE_ADDR(GADDRESS);
END
TABLE FILE WF_RETAIL_LITE
PRINT ADDRESS_LINE_1 AS Address GEOCODE1
BY POSTAL_CODE AS Zip
WHERE CITY_NAME EQ 'New York'
WHERE POSTAL_CODE FROM '10013' TO '10020'
ON TABLE SET PAGE NOPAGE
END
```

The output is shown in the following image.



Zip	Address	GEOCODE1
10013	125 Worth St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-74.00269, "y":40.71543}}
10016	139 E 35Th St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97911, "y":40.74705}}
10017	2 United Nations Plz	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97115, "y":40.75111}}
	405 E 42Nd St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.96956, "y":40.74867}}
	405 E 42Nd St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.96956, "y":40.74867}}
	219 E 42Nd St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97333, "y":40.75030}}
	330 Madison Ave	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97906, "y":40.75316}}
10018	119 W 40Th St Fl 10	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98599, "y":40.75398}}
	11 West 40Th Street	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98235, "y":40.75245}}
10019	31 West 52Nd Street	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97776, "y":40.76044}}
	1301 Ave Of The Americas	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97945, "y":40.76125}}
	1345 Avenue Of The Americas	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97843, "y":40.76264}}
	745 7Th Ave	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98340, "y":40.76077}}
10020	1221 Avenue Of The Americas	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98129, "y":40.75874}}
	1271 Avenue Of The Americas	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98018, "y":40.76025}}

GIS_GEOCODE_ADDR_CITY: Geocoding an Address Line, City, and State

GIS_GEOCODE_ADDR_CITY uses a GIS geocoding service to obtain the geometry point for an address line, city, state, and optional country. The returned value is a fixed length alphanumeric format, large enough to hold the JSON describing the geographic location (for example, A200).

Note: This function uses GIS services and requires an Esri ArcGIS adapter connection with named credentials.

Syntax:

How to Geocode an Address Line, City, and State

```
GIS_GEOCODE_ADDR_CITY( street_addr, city , state [, country])
```

where:

street_addr

Fixed length alphanumeric

Is the street address to be geocoded.

city

Fixed length alphanumeric

Is the city name associated with the street address.

state

Fixed length alphanumeric

Is the state name associated with the street address.

country

fixed length alphanumeric

Is a country name, which is optional if the country is the United States.

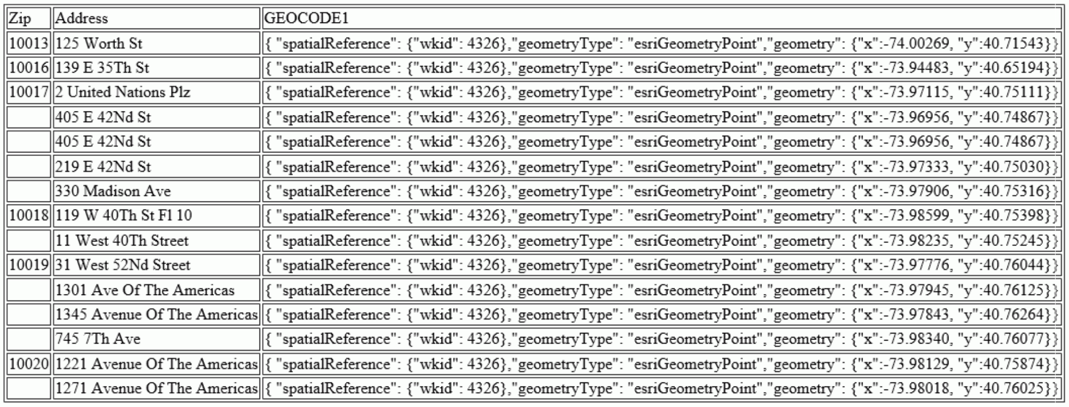
Example:

Geocoding a Street Address, City, and State

The following request geocodes a street address using GIS_GEOCODE_ADDR_CITY.

```
DEFINE FILE WF_RETAIL_LITE
GEOCODE1/A200 = GIS_GEOCODE_ADDR_CITY(ADDRESS_LINE_1, CITY_NAME ,
STATE_PROV_NAME);
END
TABLE FILE WF_RETAIL_LITE
PRINT ADDRESS_LINE_1 AS Address GEOCODE1
BY POSTAL_CODE AS Zip
WHERE CITY_NAME EQ 'New York'
WHERE POSTAL_CODE FROM '10013' TO '10020'
ON TABLE SET PAGE NOPAGE
END
```

The output is shown in the following image.



Zip	Address	GEOCODE1
10013	125 Worth St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-74.00269, "y":40.71543}}
10016	139 E 35Th St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.94483, "y":40.65194}}
10017	2 United Nations Plz	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97115, "y":40.75111}}
	405 E 42Nd St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.96956, "y":40.74867}}
	405 E 42Nd St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.96956, "y":40.74867}}
	219 E 42Nd St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97333, "y":40.75030}}
	330 Madison Ave	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97906, "y":40.75316}}
10018	119 W 40Th St Fl 10	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98599, "y":40.75398}}
	11 West 40Th Street	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98235, "y":40.75245}}
10019	31 West 52Nd Street	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97776, "y":40.76044}}
	1301 Ave Of The Americas	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97945, "y":40.76125}}
	1345 Avenue Of The Americas	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97843, "y":40.76264}}
	745 7Th Ave	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98340, "y":40.76077}}
10020	1221 Avenue Of The Americas	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98129, "y":40.75874}}
	1271 Avenue Of The Americas	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98018, "y":40.76025}}

GIS_GEOCODE_ADDR_POSTAL: Geocoding an Address Line and Postal Code

GIS_GEOCODE_ADDR_POSTAL uses a GIS geocoding service to obtain the geometry point for an address line, postal code and optional country. The returned value is a fixed length alphanumeric format, large enough to hold the JSON describing the geographic location (for example, A200).

Note: This function uses GIS services and requires an Esri ArcGIS adapter connection with named credentials.

Syntax: **How to Geocode an Address Line and Postal Code**

```
GIS_GEOCODE_ADDR_POSTAL( street_addr, postal_code [, country])
```

where:

street_addr

fixed length alphanumeric

Is the street address to be geocoded.

postal_code

fixed length alphanumeric

Is the postal code associated with the street address.

country

fixed length alphanumeric

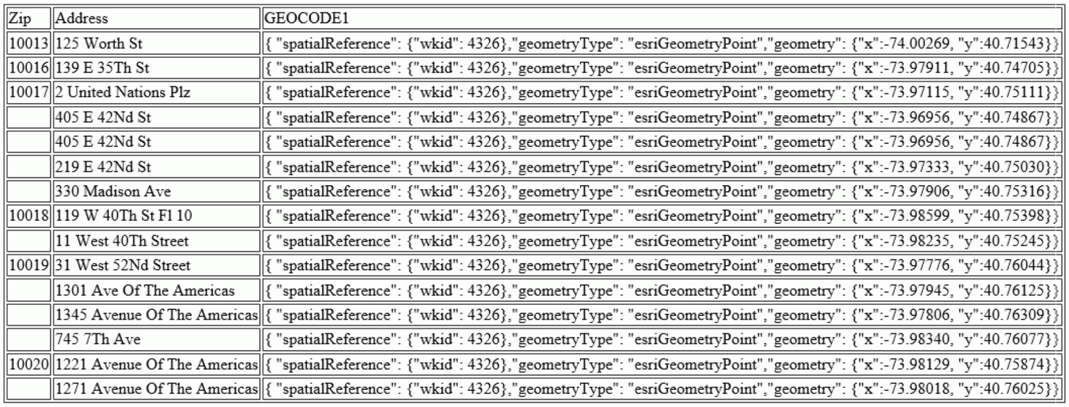
Is a country name, which is optional if the country is the United States.

Example: Geocoding a Street Address and Postal Code

The following request geocodes a street address using GIS_GEOCODE_ADDR_POSTAL.

```
DEFINE FILE WF_RETAIL_LITE
GEOCODE1/A200 = GIS_GEOCODE_ADDR_POSTAL(ADDRESS_LINE_1, POSTAL_CODE);
END
TABLE FILE WF_RETAIL_LITE
PRINT ADDRESS_LINE_1 AS Address GEOCODE1
BY POSTAL_CODE AS Zip
WHERE CITY_NAME EQ 'New York'
WHERE POSTAL_CODE FROM '10013' TO '10020'
ON TABLE SET PAGE NOPAGE
END
```

The output is shown in the following image.



Zip	Address	GEOCODE1
10013	125 Worth St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-74.00269, "y":40.71543}}
10016	139 E 35Th St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97911, "y":40.74705}}
10017	2 United Nations Plz	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97115, "y":40.75111}}
	405 E 42Nd St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.96956, "y":40.74867}}
	405 E 42Nd St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.96956, "y":40.74867}}
	219 E 42Nd St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97333, "y":40.75030}}
	330 Madison Ave	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97906, "y":40.75316}}
10018	119 W 40Th St Fl 10	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98599, "y":40.75398}}
	11 West 40Th Street	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98235, "y":40.75245}}
10019	31 West 52Nd Street	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97776, "y":40.76044}}
	1301 Ave Of The Americas	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97945, "y":40.76125}}
	1345 Avenue Of The Americas	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.97806, "y":40.76309}}
	745 7Th Ave	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98340, "y":40.76077}}
10020	1221 Avenue Of The Americas	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98129, "y":40.75874}}
	1271 Avenue Of The Americas	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.98018, "y":40.76025}}

GIS_GEOMETRY: Building a JSON Geometry Object

The GIS_GEOMETRY function builds a JSON Geometry object given a geometry type, WKID, and a geometry.

Syntax: How to Build a JSON Geometry Object

```
GIS_GEOMETRY(geotype, wkid, geometry)
```

where:

geotype

Alphanumeric

Is a geometry type, for example, 'esriGeometryPolygon' ,esriGeometryPolyline, 'esriGeometryMultipoint', 'EsriGeometryPoint', 'EsriGeometryExtent'..

wkid

Alphanumeric

Is a valid spatial reference ID. WKID is an abbreviation for Well-Known ID, which identifies a projected or geographic coordinate system.

geometry

TX

A geometry in JSON.

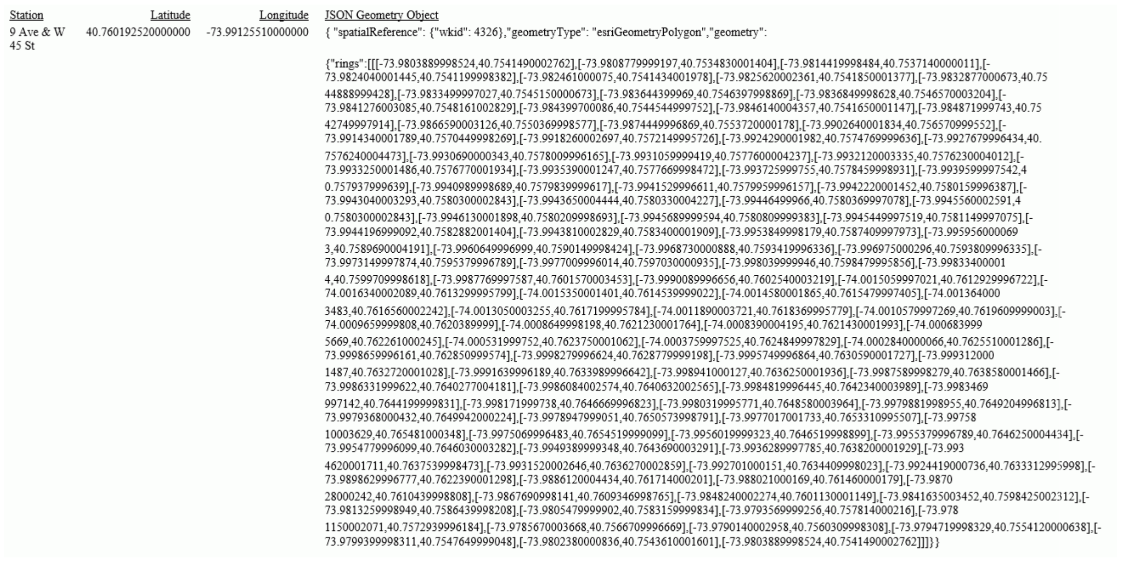
The output is returned as TX.

Example: **Building a JSON Geometry Object**

The following request builds a polygon geometry of the area encompassing ZIP code 10036 in Manhattan. The input geometry object is stored in a text (.ftm) file that is cross-referenced in the esri-citibike Master File. The field containing the geometry object is GEOMETRY.

```
DEFINE FILE esri/esri-citibike
WKID/A10  = '4326';
 MASTER_GEOMETRY/TX256 (GEOGRAPHIC_ROLE=GEOMETRY_AREA) =
    GIS_GEOMETRY( 'esriGeometryPolygon', WKID , GEOMETRY );
END
TABLE FILE esri/esri-citibike
 PRINT
    START_STATION_NAME AS Station
    START_STATION_LATITUDE AS Latitude
    START_STATION_LONGITUDE AS Longitude
    MASTER_GEOMETRY AS 'JSON Geometry Object'
 WHERE START_STATION_ID EQ 479
ON TABLE SET PAGE NOLEAD
 ON TABLE SET STYLE *
type=report, grid=off, size=10,$
 ENDSTYLE
END
```

The output is shown in the following image.



```
Station     Latitude            Longitude           JSON Geometry Object
9 Ave & W   40.760192520000000  -73.99125510000000  { "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPolygon","geometry":
45 St
                                                    {"rings":[[[-73.9803889998524,40.7541490002762],[-73.9808779999197,40.7534830001404],[-73.9814419998484,40.7537140000011],[-
                                                    73.9824040001445,40.7541199998382],[-73.982461000075,40.7541434001978],[-73.9825620002361,40.7541850001377],[-73.9832877000673,40.75
                                                    44888999428],[-73.9833499997027,40.7545150000673],[-73.983644399969,40.7546397998869],[-73.9836849998628,40.7546570003204],[-
                                                    73.9841276003085,40.7548161002829],[-73.984399700086,40.7544544999752],[-73.9846140004357,40.7541650001147],[-73.984871999743,40.75
                                                    42749997914],[-73.9866590003126,40.7550369998577],[-73.9874449996869,40.7553720000178],[-73.9902640001834,40.756570999552],[-
                                                    73.9914340001789,40.7570449998269],[-73.9918260002697,40.7572149995726],[-73.9924290001982,40.7574769999636],[-73.9927679996434,40.
                                                    7576240004473],[-73.9930690000343,40.7578009996165],[-73.9931059999419,40.7577600004237],[-73.9932120003335,40.7576230004012],[-
                                                    73.9933250001486,40.7576770001934],[-73.9935390001247,40.7577669998472],[-73.993725999755,40.7578459998931],[-73.9939599997542,4
                                                    0.757937999639],[-73.9940989998689,40.7579839999617],[-73.9941529996611,40.7579959996157],[-73.9942220001452,40.758015996387],[-
                                                    73.9943040003293,40.7580300002843],[-73.9943650004444,40.7580330004227],[-73.99446499966,40.7580369997078],[-73.9945560002591,4
                                                    0.7580300002843],[-73.9946130001898,40.7580209998693],[-73.9945689999594,40.7580809999383],[-73.9945449997519,40.7581149997075],[-
                                                    73.9944196999092,40.7582882001404],[-73.9943810002829,40.7583400001909],[-73.9953849998179,40.7587409997973],[-73.995956000069
                                                    3,40.7589690004191],[-73.9960649996999,40.7590149998424],[-73.9968730000888,40.7593419996336],[-73.996975000296,40.7593809996335],[-
                                                    73.9973149997874,40.7595379996789],[-73.9977009996014,40.7597030000935],[-73.998039999946,40.7598479995856],[-73.99833400001
                                                    4,40.7599709998618],[-73.9987769997587,40.7601570003453],[-73.9990089996656,40.7602540003219],[-74.0015059997021,40.7612929996722],[-
                                                    74.0016340002089,40.7613299995799],[-74.0015350001401,40.7614539999022],[-74.0014580001865,40.7615479997405],[-74.001364000
                                                    3483,40.7616560002242],[-74.0013050003255,40.7617199995784],[-74.0011890003721,40.7618369995779],[-74.0010579997269,40.7619609999003],[-
                                                    74.0009659999808,40.7620389999],[-74.0008649998198,40.7621230001764],[-74.0008390004195,40.7621430001993],[-74.000683999
                                                    5669,40.762261000245],[-74.000531999752,40.7623750001062],[-74.0003759997525,40.7624849997829],[-74.0002840000066,40.762551000128 6],[-
                                                    73.9998659996161,40.762850999574],[-73.9998279996624,40.7628779999198],[-73.9995749996864,40.7630590001727],[-73.999312000
                                                    1487,40.7632720001028],[-73.9991639996189,40.7633989996642],[-73.998941000127,40.7636250001936],[-73.9987589998279,40.7638580001466],[-
                                                    73.9986331999622,40.7640277004181],[-73.9986084002574,40.7640632002565],[-73.9984819996445,40.7642340003989],[-73.9983469
                                                    997142,40.7644199999831],[-73.998171999738,40.7646669996823],[-73.9980319995771,40.7648580003964],[-73.9979881998955,40.7649204996813],[-
                                                    73.9979368000432,40.7649942000224],[-73.9978947999051,40.7650573998791],[-73.9977017001733,40.7653310995507],[-73.99758
                                                    10003629,40.765481000348],[-73.9975069996483,40.7654519999099],[-73.9956019999323,40.7646519998899],[-73.9955379996789,40.7646250004434],[-
                                                    73.9954779996099,40.7646030003282],[-73.9949389999348,40.7643690003291],[-73.9936289997785,40.7638200001929],[-73.993
                                                    4620001711,40.7637539998473],[-73.9931520002646,40.7636270002859],[-73.992701000151,40.7634409998023],[-73.9924419000736,40.7633312995998],[-
                                                    73.9898629996777,40.7622390001298],[-73.9886120004434,40.761714000201],[-73.988021000169,40.761460000179],[-73.9870
                                                    28000242,40.7610439998808],[-73.9867690998141,40.7609346998765],[-73.9848240002274,40.7601130001149],[-73.9841635003452,40.7598425002312],[-
                                                    73.9813259998949,40.7586439998208],[-73.9805479999902,40.7583159999834],[-73.9793569999256,40.757814000216],[-73.978
                                                    1150002071,40.7572939996184],[-73.9785670003668,40.7566709996669],[-73.9790140002958,40.7560309998308],[-73.9794719998329,40.7554120000638],[-
                                                    73.9799399998311,40.7547649999048],[-73.9802380000836,40.7543610001601],[-73.9803889998524,40.7541490002762]]]}}
```

GIS_IN_POLYGON: Determining if a Point is in a Complex Polygon

Given a point and a polygon definition, the GIS_IN_POLYGON function returns the value 1 (TRUE) if the point is in the polygon or 0 (FALSE) if the point is not in the polygon. The value is returned in integer format.

Syntax: How to Determine if a Point is in a Complex Polygon

```
GIS_IN_POLYGON(point, polygon_definition)
```

where:

point

Alphanumeric or text

Is the geometry point.

polygon_definition

Text

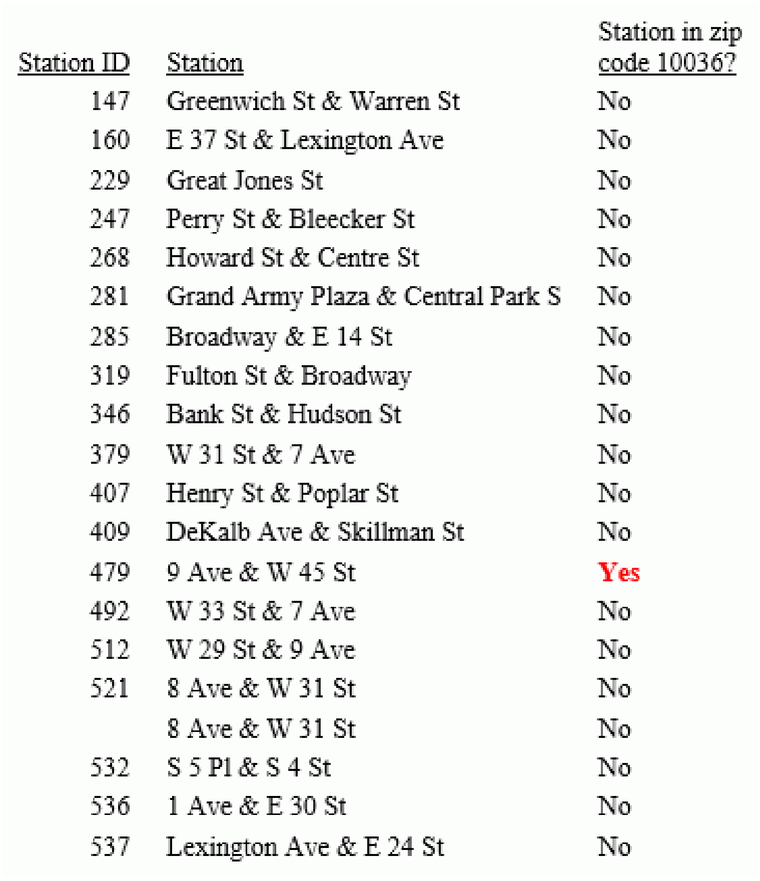
Is the geometry area (polygon) definition.

Example: Determining if a Point is in a Polygon

The following example determines if a station is inside ZIP code 10036. GIS_IN_POLYGON returns 1 for a point inside the polygon definition and 0 for a point outside. The polygon definition being passed is the same one used in the example for the GIS_GEOMETRY function described previously and defines the polygon for ZIP code 10036 in Manhattan in New York City. The value 1 is translated to Yes and 0 to No for display on the output.

```
DEFINE FILE esri/esri-citibike
WKID/A10  = '4326';
MASTER_GEOMETRY/TX256 (GEOGRAPHIC_ROLE=GEOMETRY_AREA) =
  GIS_GEOMETRY( 'esriGeometryPolygon', WKID , GEOMETRY );
START_STATION_POINT/A200=GIS_POINT(WKID, START_STATION_LONGITUDE,
START_STATION_LATITUDE);
STATION_IN_POLYGON/I4=GIS_IN_POLYGON(START_STATION_POINT, MASTER_GEOMETRY);
IN_POLYGON/A5 = IF STATION_IN_POLYGON EQ 1 THEN 'Yes' ELSE 'No';
END
TABLE FILE esri/esri-citibike
 PRINT
     START_STATION_NAME AS Station
    IN_POLYGON AS 'Station in zip, code 10036?'
BY START_STATION_ID AS 'Station ID'
ON TABLE SET PAGE NOLEAD
 ON TABLE SET STYLE *
type=report, grid=off, size=10,$
type=data, column=in_polygon, style=bold, color=red, when = in_polygon eq
'Yes',$
 ENDSTYLE
END
```

The output is shown in the following image.



Station ID	Station	Station in zip code 10036?
147	Greenwich St & Warren St	No
160	E 37 St & Lexington Ave	No
229	Great Jones St	No
247	Perry St & Bleecker St	No
268	Howard St & Centre St	No
281	Grand Army Plaza & Central Park S	No
285	Broadway & E 14 St	No
319	Fulton St & Broadway	No
346	Bank St & Hudson St	No
379	W 31 St & 7 Ave	No
407	Henry St & Poplar St	No
409	DeKalb Ave & Skillman St	No
479	9 Ave & W 45 St	Yes
492	W 33 St & 7 Ave	No
512	W 29 St & 9 Ave	No
521	8 Ave & W 31 St	No
	8 Ave & W 31 St	No
532	S 5 Pl & S 4 St	No
536	1 Ave & E 30 St	No
537	Lexington Ave & E 24 St	No

GIS_LINE: Building a JSON Line

Given two geometry points or lines, GIS_LINE builds a JSON line. The output is returned in text format.

Syntax: How to Build a JSON Line

```
GIS_LINE(geometry1, geometry2)
```

where:

geometry1

Alphanumeric or text

Is the first point or line for defining the beginning of the new line.

geometry2

Alphanumeric or text

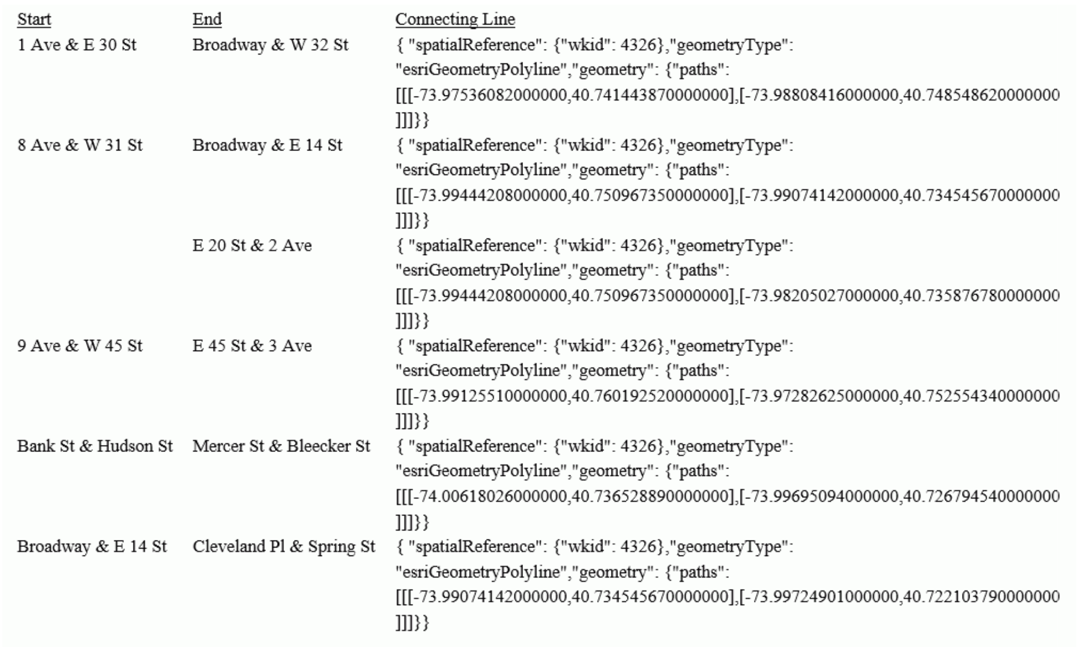
Is the second point or line for the concatenation of the new line.

Example: **Building a JSON Line**

The following request prints start stations and end stations and builds a JSON line between them.

```
DEFINE FILE ESRI/ESRI-CITIBIKE
STARTPOINT/A200 = GIS_POINT('4326', START_STATION_LONGITUDE,
START_STATION_LATITUDE);
ENDPOINT/A200 = GIS_POINT('4326', END_STATION_LONGITUDE,
END_STATION_LATITUDE);
CONNECTION_LINE/TX80 (GEOGRAPHIC_ROLE=GEOMETRY_LINE) =
   GIS_LINE(STARTPOINT, ENDPOINT);
END
TABLE FILE ESRI/ESRI-CITIBIKE
PRINT END_STATION_NAME AS End CONNECTION_LINE AS 'Connecting Line'
BY START_STATION_NAME AS Start
WHERE START_STATION_NAME LE 'D'
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
TYPE=REPORT, GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image.



Start	End	Connecting Line
1 Ave & E 30 St	Broadway & W 32 St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPolyline","geometry": {"paths": [[[-73.97536082000000,40.741443870000000],[-73.98808416000000,40.748548620000000]]]}}
8 Ave & W 31 St	Broadway & E 14 St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPolyline","geometry": {"paths": [[[-73.99444208000000,40.750967350000000],[-73.99074142000000,40.734545670000000]]]}}
	E 20 St & 2 Ave	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPolyline","geometry": {"paths": [[[-73.99444208000000,40.750967350000000],[-73.98205027000000,40.735876780000000]]]}}
9 Ave & W 45 St	E 45 St & 3 Ave	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPolyline","geometry": {"paths": [[[-73.99125510000000,40.760192520000000],[-73.97282625000000,40.752554340000000]]]}}
Bank St & Hudson St	Mercer St & Bleecker St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPolyline","geometry": {"paths": [[[-74.00618026000000,40.736528890000000],[-73.99695094000000,40.726794540000000]]]}}
Broadway & E 14 St	Cleveland Pl & Spring St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPolyline","geometry": {"paths": [[[-73.99074142000000,40.734545670000000],[-73.99724901000000,40.722103790000000]]]}}

GIS_POINT: Building a Geometry Point

Given a WKID (Well-Known ID) spatial reference, longitude, and latitude, the GIS_POINT function builds a JSON point defining a Geometry object with the provided WKID, longitude, and latitude. The function is optimized for those SQL engines that can build a JSON geometry object.

The field to which the point is returned should have fixed length alphanumeric format, large enough to hold the JSON describing the point (for example, A200).

Syntax: How to Build a Geometry Point

```
GIS_POINT(wkid, longitude, latitude)
```

where:

wkid

Fixed length alphanumeric

Is a spatial reference code (WKID). WKID is an abbreviation for Well-Known ID, which identifies a projected or geographic coordinate system.

longitude

D20.8

Is the longitude for the point.

latitude

D20.8

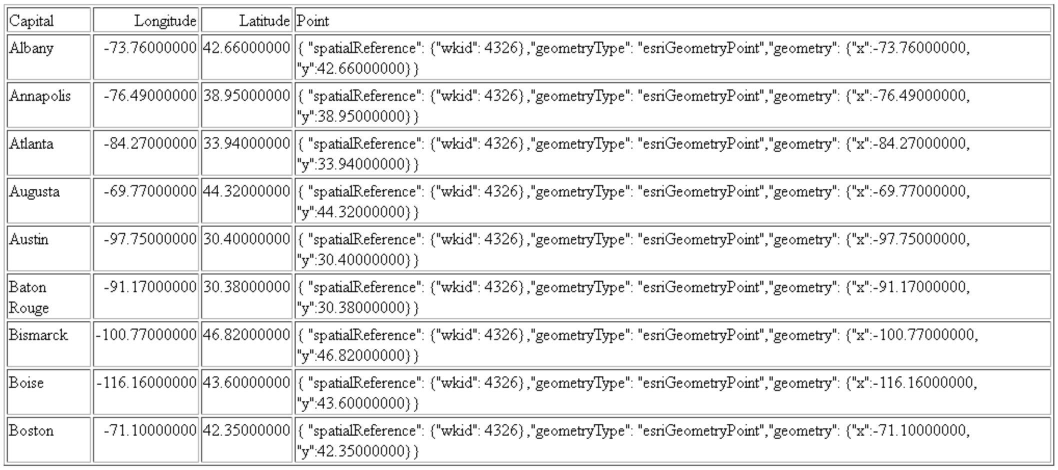
Is the latitude for the point.

Example: **Building a Geometry Point**

The following request uses the spatial reference code 4326 (decimal degrees) and state capital longitudes and latitudes to build a geometry point.

```
DEFINE FILE WF_RETAIL_LITE
GPOINT/A200 = GIS_POINT('4326', STATE_PROV_CAPITAL_LONGITUDE,
STATE_PROV_CAPITAL_LATITUDE);
END
TABLE FILE WF_RETAIL_LITE
SUM FST.STATE_PROV_CAPITAL_LONGITUDE AS Longitude
FST.STATE_PROV_CAPITAL_LATITUDE AS Latitude
FST.GPOINT AS Point
BY STATE_PROV_CAPITAL_NAME AS Capital
WHERE COUNTRY_NAME EQ 'United States'
WHERE STATE_PROV_CAPITAL_NAME LT 'C'
ON TABLE SET PAGE NOPAGE
END
```

The output is shown in the following image.



Capital	Longitude	Latitude	Point
Albany	-73.76000000	42.66000000	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-73.76000000, "y":42.66000000}}
Annapolis	-76.49000000	38.95000000	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-76.49000000, "y":38.95000000}}
Atlanta	-84.27000000	33.94000000	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-84.27000000, "y":33.94000000}}
Augusta	-69.77000000	44.32000000	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-69.77000000, "y":44.32000000}}
Austin	-97.75000000	30.40000000	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-97.75000000, "y":30.40000000}}
Baton Rouge	-91.17000000	30.38000000	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-91.17000000, "y":30.38000000}}
Bismarck	-100.77000000	46.82000000	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-100.77000000, "y":46.82000000}}
Boise	-116.16000000	43.60000000	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-116.16000000, "y":43.60000000}}
Boston	-71.10000000	42.35000000	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPoint","geometry": {"x":-71.10000000, "y":42.35000000}}

GIS_REVERSE_COORDINATE: Returning a Geographic Component

Given longitude and latitude values and the name of a geographic component, GIS_REVERSE_COORDINATE returns the specified geographic component values associated with those coordinates.

Note: This function uses GIS services and requires an Esri ArcGIS adapter connection with named credentials.

Syntax: How to Return a Geographic Component

```
GIS_REVERSE_COORDINATE(longitude, latitude, component)
```

where:

longitude

Numeric

Is the longitude of the component to return.

latitude

Numeric

Is the latitude of the component to return.

component

Keyword

Is one of the following components:

- MATCH_ADDRESS, which returns the matching address.
- METROAREA, which returns the metro area name.
- REGION, which returns the region name.
- SUBREGION, which returns the subregion name.
- CITY, which returns the city name.
- POSTAL, which returns the postal code.

The value is returned as text and can be assigned to a field with text or alphanumeric (fixed or variable length) format.

Example: Returning Geographic Components Associated With Coordinates

GIS_REVERSE_COORDINATE returns the REGION, given a city longitude and city latitude.

```
GIS_REVERSE_COORDINATE(CITY_LONGITUDE, CITY_LATITUDE, REGION)
```

For Annapolis, the result is Maryland.

For Baton Rouge, the result is Louisiana.

GIS_SERVICE_AREA: Calculating a Geometry Area Around a Given Point

The GIS_SERVICE_AREA function uses a GIS service to calculate the geometry area with access boundaries within the given time or distance from the provided geometry point. The output is returned in text format.

Note: This function uses GIS services and requires an Esri ArcGIS adapter connection with named credentials.

Syntax: How to Calculate a Geometry Area Around a Point

```
GIS_SERVICE_AREA(geo_point, distance, travel_mode)
```

where:

geo_point

Alphanumeric

Is the starting geometry point.

distance

Alphanumeric

Is the travel limitation in either time or distance units.

travel_mode

Alphanumeric

Is a valid travel mode as defined in gis_serv_area.mas in the Catalog directory under the server installation directory. The accepted travel modes are;

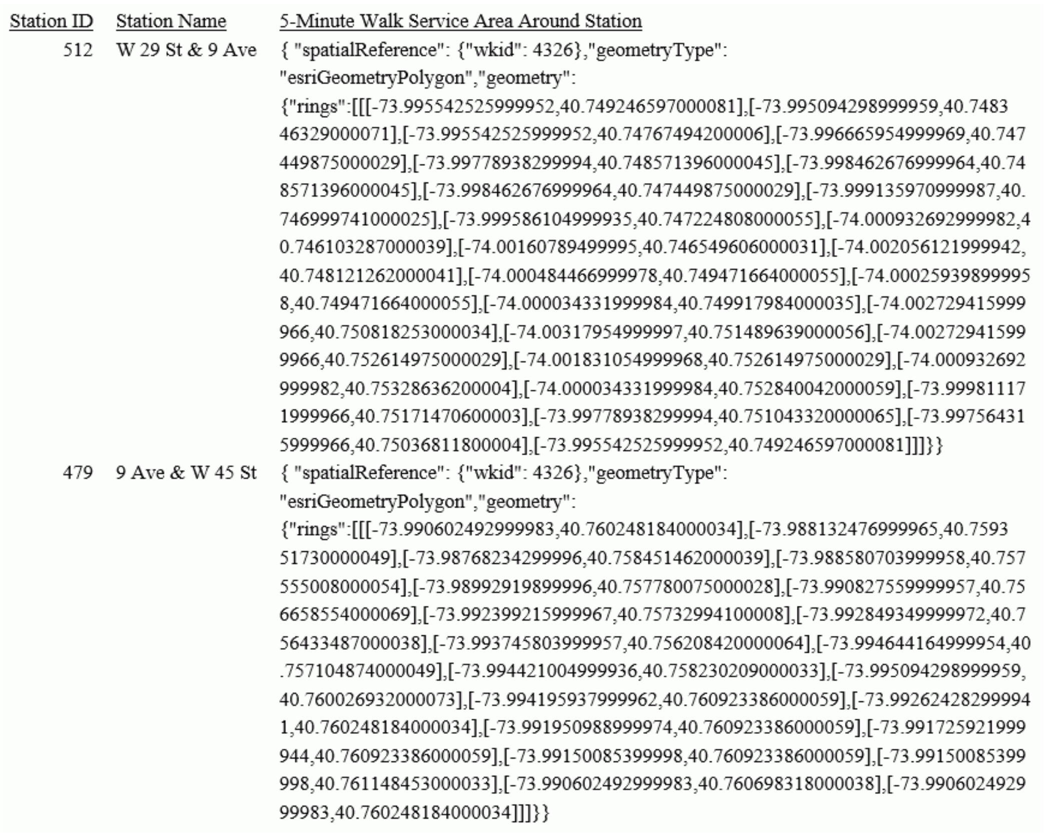
- **'Miles'**. This is the default value.
- **'TravelTime'**.
- **'TruckTravelTime'**.
- **'WalkTime'**.
- **'Kilometers'**.

Example: Calculating a Service Area Around a Geometry Point

The following request calculates the geometry area that is a five-minute walk around a station.

```
DEFINE FILE esri/esri-citibike
WKID/A10='4326';
START_STATION_POINT/A200=GIS_POINT(WKID, START_STATION_LONGITUDE,
START_STATION_LATITUDE);
DISTANCE/A10='5';
TRAVEL_MODE/A10='WalkTime';
STATION_SERVICE_AREA/TX80 (GEOGRAPHIC_ROLE=GEOMETRY_AREA)=
 GIS_SERVICE_AREA(START_STATION_POINT, DISTANCE, TRAVEL_MODE);
END
TABLE FILE esri/esri-citibike
 PRINT
    START_STATION_ID AS 'Station ID'
    START_STATION_NAME AS 'Station Name'
    STATION_SERVICE_AREA AS '5-Minute Walk Service Area Around Station'
 WHERE START_STATION_ID EQ 479 OR 512;
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
TYPE=REPORT, GRID=OFF, SIZE=12,$
ENDSTYLE
END
```

The output is shown in the following image.



Station ID	Station Name	5-Minute Walk Service Area Around Station
512	W 29 St & 9 Ave	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPolygon","geometry": {"rings":[[[-73.995542525999952,40.749246597000081],[-73.995094298999959,40.748346329000071],[-73.995542525999952,40.74767494200006],[-73.996665954999969,40.747449875000029],[-73.99778938299994,40.748571396000045],[-73.998462676999964,40.748571396000045],[-73.998462676999964,40.747449875000029],[-73.999135970999987,40.746999741000025],[-73.999586104999935,40.747224808000055],[-74.000932692999982,40.746103287000039],[-74.00160789499995,40.746549606000031],[-74.002056121999942,40.748121262000041],[-74.000484466999978,40.749471664000055],[-74.000259398999958,40.749471664000055],[-74.000034331999984,40.749917984000035],[-74.002729415999966,40.750818253000034],[-74.00317954999997,40.751489639000056],[-74.002729415999966,40.752614975000029],[-74.001831054999968,40.752614975000029],[-74.000932692999982,40.75328636200004],[-74.000034331999984,40.752840042000059],[-73.999811171999966,40.75171470600003],[-73.99778938299994,40.751043320000065],[-73.997564315999966,40.75036811800004],[-73.995542525999952,40.749246597000081]]]}}
479	9 Ave & W 45 St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPolygon","geometry": {"rings":[[[-73.990602492999983,40.760248184000034],[-73.988132476999965,40.759351730000049],[-73.98768234299996,40.758451462000039],[-73.988580703999958,40.757555008000054],[-73.98992919899996,40.757780075000028],[-73.990827559999957,40.756658554000069],[-73.992399215999967,40.75732994100008],[-73.992849349999972,40.756433487000038],[-73.993745803999957,40.756208420000064],[-73.994644164999954,40.757104874000049],[-73.994421004999936,40.758230209000033],[-73.995094298999959,40.760026932000073],[-73.994195937999962,40.760923386000059],[-73.992624282999941,40.760248184000034],[-73.991950988999974,40.760923386000059],[-73.991725921999944,40.760923386000059],[-73.99150085399998,40.760923386000059],[-73.991500853999998,40.761148453000033],[-73.990602492999983,40.760698318000038],[-73.990602492999983,40.760248184000034]]]}}

GIS_SERV_AREA_XY: Calculating a Service Area Around a Given Coordinate

The GIS_SERV_AREA_XY function uses a GIS service to calculate the geometry area with access boundaries within the given time or distance from the provided coordinate. The output is returned in text format.

Note: This function uses GIS services and requires an Esri ArcGIS adapter connection with named credentials.

Syntax: **How to Calculate a Geometry Area Around a Coordinate**

```
GIS_SERV_AREA_XY(longitude, latitude, distance, travel_mode[, wkid])
```

where:

`longitude`

Alphanumeric

Is the longitude of the starting point.

`latitude`

Alphanumeric

Is the latitude of the starting point.

`distance`

Integer

Is the travel limitation in either time or distance units.

`travel_mode`

Alphanumeric

Is a valid travel mode as defined in gis_serv_area.mas in the Catalog directory under the server installation directory. The accepted travel modes are;

- **'Miles'**. This is the default value.
- **'TravelTime'**.
- **'TruckTravelTime'**.
- **'WalkTime'**.
- **'Kilometers'**.

`wkid`

Alphanmeric

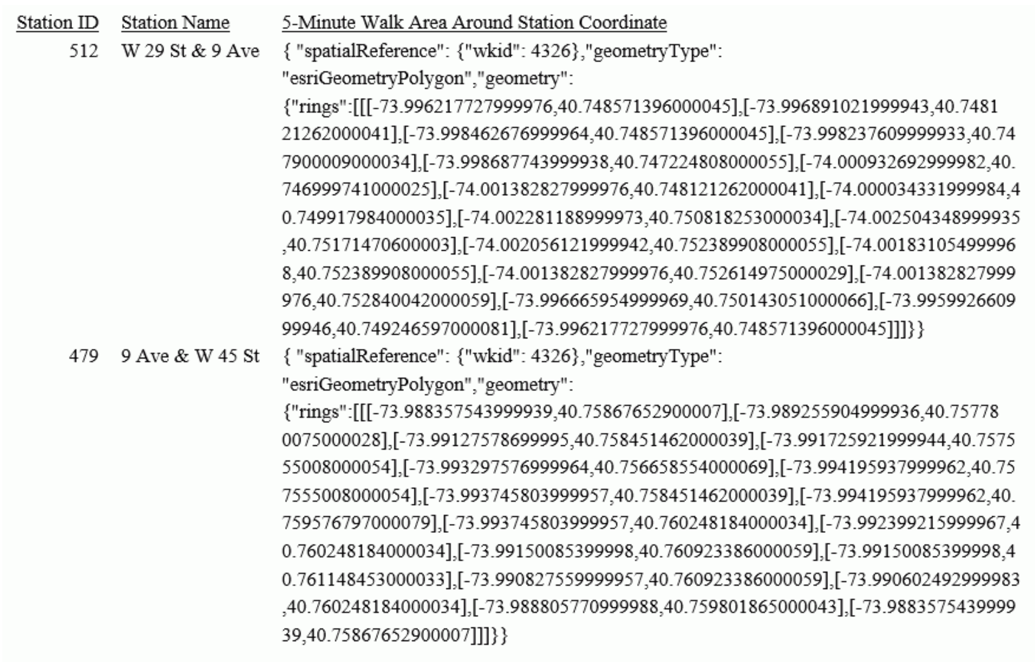
Is the spatial reference ID for the coordinate. WKID is an abbreviation for Well-Known ID, which identifies a projected or geographic coordinate system. The default value is '4326', which represents decimal degrees.

Example: Calculating a Service Area Around a Coordinate

The following request calculates the geometry area that is a five-minute walk around a station, using the longitude and latitude that specify the station location.

```
DEFINE FILE esri/esri-citibike
DISTANCE/I4=5;
WKID/A10='4326';
TRAVEL_MODE/A10='WalkTime';
STATION_SERVICE_AREA/TX80 (GEOGRAPHIC_ROLE=GEOMETRY_AREA)=
   GIS_SERV_AREA_XY(START_STATION_LONGITUDE, START_STATION_LATITUDE,
DISTANCE, TRAVEL_MODE, WKID);
END
TABLE FILE esri/esri-citibike
 PRINT
    START_STATION_ID AS 'Station ID'
    START_STATION_NAME AS 'Station Name'
    STATION_SERVICE_AREA
       AS '5-Minute Walk Service Area Around Station Coordinate'
 WHERE START_STATION_ID EQ 479 OR 512;
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
TYPE=REPORT, GRID=OFF, SIZE=12,$
ENDSTYLE
END
```

The output is shown in the following image.



Station ID	Station Name	5-Minute Walk Area Around Station Coordinate
512	W 29 St & 9 Ave	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPolygon","geometry": {"rings":[[[-73.996217727999976,40.748571396000045],[-73.996891021999943,40.748121262000041],[-73.998462676999964,40.748571396000045],[-73.998237609999933,40.747900009000034],[-73.998687743999938,40.747224808000055],[-74.000932692999982,40.746999741000025],[-74.001382827999976,40.748121262000041],[-74.000034331999984,40.749917984000035],[-74.002281188999973,40.750818253000034],[-74.002504348999935,40.75171470600003],[-74.002056121999942,40.752389908000055],[-74.001831054999968,40.752389908000055],[-74.001382827999976,40.752614975000029],[-74.001382827999976,40.752840042000059],[-73.996665954999969,40.750143051000066],[-73.995992660999946,40.749246597000081],[-73.996217727999976,40.748571396000045]]]}}
479	9 Ave & W 45 St	{ "spatialReference": {"wkid": 4326},"geometryType": "esriGeometryPolygon","geometry": {"rings":[[[-73.988357543999939,40.75867652900007],[-73.989255904999936,40.757780075000028],[-73.99127578699995,40.758451462000039],[-73.991725921999944,40.757555008000054],[-73.993297576999964,40.756658554000069],[-73.994195937999962,40.757555008000054],[-73.993745803999957,40.758451462000039],[-73.994195937999962,40.759576797000079],[-73.993745803999957,40.760248184000034],[-73.992399215999967,40.760248184000034],[-73.99150085399998,40.760923386000059],[-73.99150085399998,40.761148453000033],[-73.990827559999957,40.760923386000059],[-73.990602492999983,40.760248184000034],[-73.988805770999988,40.759801865000043],[-73.988357543999939,40.75867652900007]]]}}

Chapter 19

SQL Character Functions

SQL character functions manipulate alphanumeric fields and character strings.

In this chapter:

CHAR_LENGTH: Finding the Length of a Character String

The CHAR_LENGTH function returns the length of a character string. CHARACTER_LENGTH is identical to CHAR_LENGTH.

This function is most useful for columns described as VARCHAR (variable length character). For example, if a column described as GLOSS VARCHAR(10) contains

```
'bryllig'
'slythy '
'toves  '
```

then CHAR_LENGTH(GLOSS) would return

7
6
5

If the column is described as CHAR (non-variable length character), the same number is returned for all rows. In this case, CHAR_LENGTH(GLOSS) would return

10
10
10

To avoid counting trailing blanks use CHAR_LENGTH(TRIM (TRAILING FROM GLOSS)). See *TRIM: Removing Leading or Trailing Characters (SQL)* on page 418 for details.

Syntax: How to Find the Length of a Character String

```
CHAR_LENGTH(arg)
```

where:

arg

Character string

Is the value whose length is to be determined.

This function returns an integer value.

Example: Finding the Length of a Character String

CHAR_LENGTH finds the length of the string. This example,

```
CHAR_LENGTH('abcdef')
```

returns 6.

This example,

```
CHAR_LENGTH('abcdef   ')
```

returns 9, since trailing blanks are counted.

CONCAT: Concatenating Two Character Strings

The CONCAT function concatenates the values of two arguments. The result is a character string consisting of the characters of the first argument followed by the characters of the second argument.

Syntax: How to Concatenate Two Character Strings

```
CONCAT(arg1, arg2)
```

where:

arg1, arg2

Character strings

Are the strings to be concatenated.

The length of the result is the sum of the lengths of the two arguments. If either argument is variable-length, so is the result; otherwise, the result is fixed-length.

Example: Concatenating Two Character Strings

CONCAT concatenates two string. This example,

```
CONCAT('abc', 'def')
```

returns abcdef.

DIFFERENCE: Measuring the Phonetic Similarity Between Character Strings

DIFFERENCE returns an integer value measuring the difference between the SOUNDEX or METAPHONE values of two character expressions.

Syntax: How to Measure the Phonetic Similarity Between Character String

```
DIFFERENCE(chrexp1, chrexp2)
```

where:

`chrexp1, chrexp2`

Alphanumeric

Are the character strings to be compared.

Zero (0) represents the least similarity. For SOUNDEX, 4 represents the most similarity, and for METAPHONE, 16 represents the most similarity.

The use of SOUNDEX or METAPHONE depends on the PHONETIC_ALGORITHM setting. METAPHONE is the default algorithm.

Example: Measuring the Phonetic Similarity Between Character Strings

DIFFERENCE compares the character strings *Green* and *Greene*.

```
DIFFERENCE('Green','Greene')
```

For the phonetic algorithm METAPHONE (the default), the result is 16.

EDIT: Editing a Value According to a Format (SQL)

The EDIT function edits a numeric or character value according to a format specified by a mask. (It works exactly like the EDIT function in FOCUS.)

A 9 in the mask indicates the corresponding character in the source value is copied into the result. A $ in the mask indicates that the corresponding character is to be ignored. Any other character is inserted into the result.

Syntax: How to Edit a Value According to a Format

```
EDIT(arg, mask)
```

where:

`arg`

Numeric or character string

Is the value to be edited.

mask

character string

Indicates how the editing is to proceed.

This function returns a character string whose length is determined by the mask.

Example: Editing a Value According to a Format

EDIT extracts a character from a string. This example,

```
EDIT('FRED'        , '9$$$')
```

returns F.

This example,

```
EDIT('123456789', '999-99-9999')
```

returns 123-45-6789.

GET_TOKEN: Extracting a Token Based on a String of Delimiters

GET_TOKEN extracts a token (substring) based on a string that can contain multiple characters, each of which represents a single-character delimiter.

Syntax: How to Extract a Token Based on a String of Delimiters

```
GET_TOKEN(string, delimiter_string, occurrence)
```

where:

string

Alphanumeric

Is the input string from which the token will be extracted. This can be an alphanumeric field or constant.

delimiter_string

Alphanumeric constant

Is a string that contains the list of delimiter characters. For example, '; ,' contains three delimiter characters, semi-colon, blank space, and comma.

occurrence

Integer constant

Is a positive integer that specifies the token to be extracted. A negative integer will be accepted in the syntax, but will not extract a token. The value zero (0) is not supported.

Example: Extracting a Token Based on a String of Delimiters

GET_TOKEN extracts a token based on a string of delimiters.

```
GET_TOKEN(InputString, ',;/', 4)
```

For input string 'ABC,DEF;GHI/JKL', the result is JKL.

INITCAP: Capitalizing the First Letter of Each Word in a String

INITCAP capitalizes the first letter of each word in an input string and makes all other letters lowercase. A word starts at the beginning of the string, after a blank space, or after a special character.

Syntax: How to Capitalize the First Letter of Each Word in a String

```
INITCAP(input_string)
```

where:

input_string

Alphanumeric

Is the string to capitalize.

Example: Capitalizing the First Letter of Each Word in a String

INITCAP capitalizes the first letter of each word.

```
INITCAP(NewName)
```

For the string abc,def!ghi'jKL MNO, the result is Abc,Def!Ghi'Jkl Mno.

For MCKNIGHT, the result is Mcknight.

LCASE: Converting a Character String to Lowercase

The LCASE function converts a character string value to lowercase. That is, capital letters are replaced by their corresponding lowercase values.

LOWER and LOWERCASE are identical to LCASE.

Syntax: How to Convert a Character String to Lowercase

```
LCASE(arg)
```

where:

`arg`

Character string

Is the value to be converted to lowercase.

This function returns a varying character string. The length is the same as the input argument.

Example: Converting a Character String to Lowercase

LCASE converts a character string to lowercase. This example,

```
LCASE('XYZ')
```

returns xyz.

LEFT: Returning Characters From the Left of a Character String

Given a source character string, or an expression that can be converted to varchar (variable-length alphanumeric), and an integer number, LEFT returns that number of characters from the left end of the string.

Syntax: How to Return Characters From the Left of a Character String

```
LEFT(chr_exp, int_exp)
```

where:

`chr_exp`

Alphanumeric or an expression that can be converted to variable-length alphanumeric.

Is the source character string.

`int_exp`

Integer

Is the number of characters to be returned.

Example: Returning Characters From the Left of a Character String

LEFT returns the two leftmost characters from SOURCE:

```
LEFT(SOURCE,2)
```

For 'abcdefg', the result is *ab*.

LIKE: Filtering Using a Mask

LIKE accepts strings that conform to a mask. A mask is an alphanumeric pattern that you supply for comparison to characters in a data field. The data field must have an alphanumeric format (A).

Syntax: **How to Filter Using a Mask**

```
string LIKE 'mask' [ESCAPE 'c']
```

where:

string
: Is a field containing the input string or the input string enclosed in single quotation marks.

'*mask*'
: Is an alphanumeric or text character string you supply. There are two wildcard characters that you can use in the mask. The underscore (_) indicates that any character in that position is acceptable, and the percent sign (%) allows any following sequence of zero or more characters.

'*c*'
: Is any single character that you identify as the escape character. If you embed the escape character in the mask, before a percent sign (%) or underscore (_), the % or _ character is treated as a literal, rather than as a wildcard. The single quotation marks are required.

Example: **Using LIKE**

To search for the first name with the characters 'Abel' plus one additional character, you can issue the following filter:

```
FIRSTNAME LIKE 'Abel_'
```

The strings 'Abela' and 'Abele' will pass the test.

To search for first names of any length that start with the characters 'Abel', you can issue the following filter:

```
FIRSTNAME LIKE 'Abel%'
```

The strings 'Abel', 'Abela', 'Abelard', 'Abelina', and 'Abele' will pass the test.

LOCATE: Returning the Position of a Substring in a String

Given a substring, a source string and a starting position (the default is 1), LOCATE returns the position of the first occurrence of the substring, starting the search at the starting position. If the substring is not found, LOCATE returns zero (0). The search is case insensitive.

Syntax: How to Return the Position of a Substring in a String

```
LOCATE(substr, source [,start])
```

where:

substr

Alphanumeric

Is the search string.

source

Alphanumeric

Is the source string.

start

Numeric

Is the optional starting position for the search. If omitted, it defaults to 1.

Example: Returning the Position of a Substring in a String

LOCATE searches for the substring 'a' in CustomerName, starting the search in position 3.

```
LOCATE('a', CustomerName, 3)
```

For Sandra Arzola, the result 6:

LPAD: Left-Padding a Character String

LPAD uses a specified character and output length to return a character string padded on the left with that character.

Syntax: How to Pad a Character String on the Left

```
LPAD(string, out_length, pad_character)
```

where:

`string`

Fixed length alphanumeric

Is a string to pad on the left side.

`out_length`

Integer

Is the length of the output string after padding.

`pad_character`

Fixed length alphanumeric

Is a single character to use for padding.

Example: **Left-Padding a String**

LPAD left-pads the PRODUCT_CATEGORY column with @ symbols:

```
LPAD(PRODUCT_CATEGORY,25,'@')
```

For *Stereo Systems*, the output is @@@@@@@@@@@*Stereo Systems*.

Reference: **Usage Notes for LPAD**

- To use the single quotation mark (') as the padding character, you must double it and enclose the two single quotation marks within single quotation marks (LPAD(COUNTRY, 20,''''). You can use an amper variable in quotation marks for this parameter, but you cannot use a field, virtual or real.
- Input can be fixed or variable length alphanumeric.
- Output, when optimized to SQL, will always be data type VARCHAR.
- If the output is specified as shorter than the original input, the original data will be truncated, leaving only the padding characters. The output length can be specified as a positive integer or an unquoted &variable (indicating a numeric).

LTRIM: Removing Leading Spaces

The LTRIM function removes leading spaces from a character string.

Syntax: **How to Remove Leading Spaces**

```
LTRIM(arg)
```

where:

arg

character string

Is the value to be trimmed.

This function returns a varying character string. The data type of the result has a length equal to that of the input argument (although the value may be shorter).

Example: Removing Leading Spaces

LTRIM removes leading spaces. This example,

```
LTRIM('   ABC   ')
```

returns 'ABC   '.

OVERLAY: Replacing Characters in a String

Given a starting position, length, source string, and insertion string, OVERLAY replaces the number of characters defined by *length* in the source string with the insertion string, starting from the starting position.

Syntax: How to Replace Characters in a String

```
OVERLAY(src, ins, start, len)
```

where:

src

Alphanumeric

Is the source string whose characters will be replaced.

ins

Alphanumeric

Is the insertion string with the replacement characters.

start

Numeric

Is the starting position for the replacement in the source string.

len

Numeric

Is the number of characters to replace in the source string with the entire insertion string.

Example: **Replacing Characters in a String**

OVERLAY replaces the first three characters in 'ENGLAND' with the characters 'SCOT'.

```
OVERLAY('ENGLAND', 'SCOT', 1, 3)
```

The result is 'SCOTLAND'.

PATTERNS: Returning a Pattern That Represents the Structure of the Input String

PATTERNS returns a string that represents the structure of the input argument. The returned pattern includes the following characters:

- **A** is returned for any position in the input string that has an uppercase letter.
- **a** is returned for any position in the input string that has a lowercase letter.
- **9** is returned for any position in the input string that has a digit.

Note that special characters (for example, +-/=%) are returned exactly as they were in the input string.

The output is returned as variable length alphanumeric.

Syntax: **How to Return a String That Represents the Pattern Profile of the Input Argument**

```
PATTERNS(string)
```

where:

string

Alphanumeric

Is a string whose pattern will be returned.

Example: **Returning a Pattern Representing an Input String**

PATTERNS returns the pattern representing the field ADDRESS_LINE_1.

```
PATTERNS(ADDRESS_LINE_1)
```

For 1010 Milam St # Ifp-2352

The result is 9999 Aaaaa Aa # Aaa-9999.

POSITION: Finding the Position of a Substring

The POSITION function returns the position within a character string of a specified substring. If the substring does not appear in the character string, the result is 0. Otherwise, the value returned is one greater than the number of characters in the string preceding the start of the first occurrence of the substring.

Syntax: How to Find the Position of a Substring

```
POSITION(substring IN arg)
```

where:

`substring`

character string

Is the substring to search for.

`arg`

character string

Is the string to be searched for the substring.

This function returns an integer value.

Example: Finding the Position of a Substring

POSITION returns the position of a substring. This example,

```
POSITION ('A'    IN 'AEIOU')
```

returns 1.

This example,

```
POSITION ('IOU' IN 'AEIOU')
```

returns 3.

This example,

```
POSITION ('Y'     IN 'AEIOU')
```

returns 0.

POSITION: Returning the Position of a Search String in a Source String

Given a search string, a source string, and a starting position, POSITION returns the position of the search string within the source string. The search starts at the given starting position and searches from left to right. If the string is not found, POSITION returns zero (0). The search is case sensitive.

Syntax: How to Return the Position of a Search String in a Source String

```
POSITION(search, source, start)
```

where:

search

Alphanumeric

Is the search string.

source

Alphanumeric

Is the source string.

start

Numeric

Is the starting position in the source string for the search.

Example: Returning the Position of a Search String in a Source String

POSITION finds the first instance of the uppercase letter A in CustomerName after position 3.

```
POSITION('A', CustomerName, 3)
```

For *Sandra Arzola*, the result is 8.

REGEXP_COUNT: Counting the Number of Matches to a Pattern in a String

REGEXP_COUNT returns the integer count of matches to a specified regular expression pattern within a source string.

Syntax: How to Count the Number of Matches to a Pattern in a String

```
REGEXP_COUNT(string, pattern)
```

where:

string

Alphanumeric

Is the input string to be searched.

pattern

Alphanumeric

Is a regular expression, enclosed in single quotation marks, constructed using literals and meta-characters. The following meta-characters are supported

- . represents any single character
- * represents zero or more occurrences
- + represents one or more occurrences
- ? represents zero or one occurrence
- ^ represents beginning of line
- $ represents end of line
- [] represents any one character in the set listed within the brackets
- [^] represents any one character not in the set listed within the brackets
- | represents the Or operator
- \ is the Escape Special Character
- () contains a character sequence

Example: Counting the Number of Matches to a Pattern in a String

The following examples use the following Regular Expression symbols.

- $, which searches for a specified expression that occurs at the end of a string.
- ^, which searches for a specified expression that occurs at the beginning of a string.

REGEXP_COUNT counts the number of occurrences of the characters 'umpty' that occur at the end of the string 'Humpty Dumpty'.

```
REGEXP_COUNT('Humpty Dumpty', 'umpty$')
```

The result is 1.

REGEXP_COUNT counts the number of occurrences of the characters 'umpty' that occur at the beginning of the string 'Humpty Dumpty'.

```
REGEXP_COUNT('Humpty Dumpty', '^umpty')
```

The result is 0.

REGEXP_INSTR: Returning the First Position of a Pattern in a String

REGEXP_INSTR returns the integer position of the first match to a specified regular expression pattern within a source string. The first character position in a string is indicated by the value 1. If there is no match within the source string, the value 0 is returned.

Syntax: **How to Return the Position of a Pattern in a String**

```
REGEXP_INSTR(string, pattern)
```

where:

string

Alphanumeric

Is the input string to be searched.

pattern

Alphanumeric

Is a regular expression, enclosed in single quotation marks, constructed using literals and meta-characters. The following meta-characters are supported

- . represents any single character
- * represents zero or more occurrences
- + represents one or more occurrences
- ? represents zero or one occurrence
- ^ represents beginning of line
- $ represents end of line
- [] represents any one character in the set listed within the brackets
- [^] represents any one character not in the set listed within the brackets
- | represents the Or operator
- \ is the Escape Special Character
- () contains a character sequence

Example: Finding the Position of a Pattern in a String

The following examples use the following Regular Expression symbols.

- ❑ $, which searches for a specified expression that occurs at the end of a string.
- ❑ ^, which searches for a specified expression that occurs at the beginning of a string.

REGEXP_INSTR finds the position of the characters 'umpty' that occur at the end of the string 'Humpty Dumpty'.

```
REGEXP_INSTR('Humpty Dumpty', 'umpty$')
```

The result is 9.

REGEXP_INSTR finds the position of the characters 'umpty' that occur at the beginning of the string 'Humpty Dumpty'.

```
REGEXP_INSTR('Humpty Dumpty', '^umpty')
```

The result is 0.

REGEXP_REPLACE: Replacing All Matches to a Pattern in a String

REGEXP_REPLACE returns a string generated by replacing all matches to a regular expression pattern in the source string with the given replacement string. The replacement string can be a null string.

Syntax: How to Replace Matches to a Pattern in a String

```
REGEXP_REPLACE(string, pattern, replacement)
```

where:

string

Alphanumeric

Is the input string to be searched.

pattern

Alphanumeric

Is a regular expression, enclosed in single quotation marks, constructed using literals and meta-characters. The following meta-characters are supported

- ❑ . represents any single character
- ❑ * represents zero or more occurrences

- + represents one or more occurrences
- ? represents zero or one occurrence
- ^ represents beginning of line
- $ represents end of line
- [] represents any one character in the set listed within the brackets
- [^] represents any one character not in the set listed within the brackets
- | represents the Or operator
- \ is the Escape Special Character
- () contains a character sequence

`replacement`

Alphanumeric

Is the replacement string.

Example: **Replacing Matches to a Pattern in a String**

The following example uses the following Regular Expression symbol.

- ^, which searches for a specified expression that occurs at the beginning of a string.

REGEXP_REPLACE replaces the characters 'ENG' at the beginning of the field COUNTRY with the replacement string 'SCOT'.

```
REGEXP_REPLACE(COUNTRY, '^ENG', 'SCOT')
```

For 'ENGLAND', the result is 'SCOTLAND'.

REGEXP_SUBSTR: Returning the First Match to a Pattern in a String

REGEXP_SUBSTR returns a string that contains the first match to a specified regular expression pattern within a source string. If there is no match within the source string, a null string is returned.

Syntax: **How to Returning the First Match to a Pattern in a String**

```
REGEXP_SUBSTR(string, pattern)
```

where:

`string`

Alphanumeric

Is the input string to be searched.

`pattern`

Alphanumeric

Is a regular expression, enclosed in single quotation marks, constructed using literals and meta-characters. The following meta-characters are supported

- . represents any single character
- * represents zero or more occurrences
- + represents one or more occurrences
- ? represents zero or one occurrence
- ^ represents beginning of line
- $ represents end of line
- [] represents any one character in the set listed within the brackets
- [^] represents any one character not in the set listed within the brackets
- | represents the Or operator
- \ is the Escape Special Character
- () contains a character sequence

Example: Returning the First Match of a Pattern in a String

The following example uses the following Regular Expression symbols.

- [A-Z], which matches any uppercase letter.
- $, which searches for a specified expression that occurs at the end of a string.

REGEXP_SUBSTR searches for a string with any uppercase letter followed by the characters 'umpty' at the end of the string 'Humpty Dumpty'.

```
REGEXP_SUBSTR('Humpty Dumpty', '[A-Z]umpty$')
```

The result is 'Dumpty'.

REPEAT: Repeating a String a Given Number of Times

Given a source string and an integer number, REPEAT returns a string with the source string repeated that number of times. The string containing the repeated strings must be large enough to fit the repetitions or it will contain a truncated value.

Syntax: How to Repeat a Character String a Given Number of Times

```
REPEAT(source_str, number)
```

where:

source_str
: Alphanumeric

Is the source string to be repeated. If *source_str* is a field, the entire field, including blanks, will be repeated.

number
: Numeric

Is the number of times to repeat the source string.

Example: Repeating a String a Given Number of Times

REPEAT returns a string with FIRST_NAME repeated three times.

```
REPEAT(FIRST_NAME, 3)
```

For MARY, the result is *MARY MARY MARY*.

REPLACE: Replacing a String

REPLACE replaces all instances of a search string in an input string with the given replacement string. The output is always variable length alphanumeric with a length determined by the input parameters.

Syntax: How to Replace all Instances of a String

```
REPLACE(input_string , search_string , replacement)
```

where:

input_string
: Alphanumeric or text (An, AnV, TX)

Is the input string.

search_string

Alphanumeric or text (An, AnV, TX)

Is the string to search for within the input string.

replacement

Alphanumeric or text (An, AnV, TX)

Is the replacement string to be substituted for the search string. It can be a null string ('').

Example: **Replacing a String**

REPLACE replaces the string 'South' in the Country Name with the string 'S.'

```
REPLACE(COUNTRY_NAME, 'SOUTH', 'S.');
```

For South Africa, the result is S. Africa.

Example: **Replacing All Instances of a String**

REPLACE removes the characters 'DAY' from the string DAY1:

```
REPLACE(DAY1, 'DAY', '' )
```

For 'SUNDAY MONDAY TUESDAY', the result is 'SUN MON TUES'.

REVERSE: Reversing the Characters in a String

REVERSE reverses the characters in a string.

Syntax: **How to Reverse the Characters in a String**

```
REVERSE(string)
```

where:

string

Is the field containing the string, or the string enclosed in single quotation marks.

Example: **Reversing the Characters in a String**

REVERSE reverses the characters in PRODCAT:

```
REVERSE(PRODCAT)
```

For VCRs, the result is sRCV.

For DVD, the result is DVD.

RIGHT: Returning the Right Portion of a String

RIGHT returns the number of characters specified starting from the end of a string and moving forward.

Syntax: How to Return the Right Portion of a String

```
RIGHT(source_string, substring_limit)
```

where:

source_string
Is a field containing the source string, or the string enclosed in single quotation marks.

substring_limit
Is the integer number of characters in the substring to be returned.

Example: Returning the Right Portion of a String

RIGHT('ABC', 2) returns 'BC'.

RIGHT('ABC', 1) returns 'C'.

RLIKE: Filtering Using a Regular Expression

RLIKE compares a string to a regular expression and accepts values that conform to the regular expression. A regular expression is a sequence of characters that define a search pattern. For complete information about regular expressions, you can search online.

Syntax: How to Filter Using a Regular Expression

```
string RLIKE 'regex'
```

where:

string
Is a field containing the input string, or the input string enclosed in single quotation marks.

'*regex*'
Is the regular expression to be used for comparison, enclosed in single quotation marks.

Example: Filtering Using a Regular Expression

FIRST_NAME RLIKE '^Ste(v|ph)en$' returns TRUE for field FIRST_NAME values starting with "Ste" followed by either "ph" or "v" and ending with "en".

RPAD: Right-Padding a Character String

RPAD uses a specified character and output length to return a character string padded on the right with that character.

Syntax: **How to Pad a Character String on the Right**

```
RPAD(string, out_length, pad_character)
```

where:

string

Alphanumeric

Is a string to pad on the right side.

out_length

Integer

Is the length of the output string after padding.

pad_character

Alphanumeric

Is a single character to use for padding.

Example: **Right-Padding a String**

RPAD right-pads the PRODUCT_CATEGORY column with @ symbols:

```
RPAD(PRODUCT_CATEGORY,25,'@')
```

For *Stereo Systems*, the output is *Stereo Systems@@@@@@@@@@@*.

Reference: **Usage Notes for RPAD**

- ❏ The input string can be data type AnV, VARCHAR, TX, and An.
- ❏ Output can only be AnV or An.
- ❏ When working with relational VARCHAR columns, there is no need to trim trailing spaces from the field if they are not desired. However, with An and AnV fields derived from An fields, the trailing spaces are part of the data and will be included in the output, with the padding being placed to the right of these positions. You can use TRIM or TRIMV to remove these trailing spaces prior to applying the RPAD function.

RTRIM: Removing Trailing Spaces

The RTRIM function removes trailing spaces from a character string.

Syntax: **How to Remove Trailing Spaces**

```
RTRIM(arg)
```

where:

arg

character string

Is the value to be trimmed.

This function returns a varying character string. The data type of the result has a length equal to that of the input argument (although the value may be shorter).

Example: **Removing Trailing Spaces**

RTRIM removes trailing spaces. This example,

```
RTRIM('   ABC   ')
```

returns '   ABC'.

SPACE: Returning a String With a Given Number of Spaces

Given an integer count, SPACE returns a string consisting of that number of spaces.

Note: To retain the spaces in HTML report output, the SHOWBLANKS parameter must be set to ON.

Syntax: **How to Return a String With a Given Number of Spaces**

```
SPACE(count)
```

where:

count

Numeric

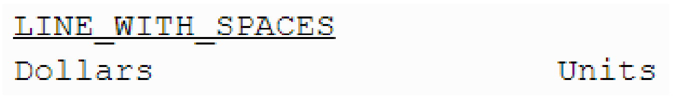
Is the number of spaces to return.

Example: Returning a String With a Given Number of Spaces

SPACE adds 20 blank spaces between the words 'Dollars' and 'Units' using the monospaced Courier font.

```
SET SHOWBLANKS = ON
SQL
SELECT
('Dollars' || SPACE(20) || 'Units') AS LINE_WITH_SPACES ;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF, FONT=COURIER,$
ENDSTYLE
END
```

The output is shown in the following image.



```
LINE_WITH_SPACES
Dollars                    Units
```

SPLIT: Extracting an Element From a String

The SPLIT function returns a specific type of element from a string. The output is returned as variable length alphanumeric.

Syntax: How to Extract an Element From a String

```
SPLIT(element, string)
```

where:

element

Can be one of the following keywords:

- **EMAIL_DOMAIN.** Is the domain name portion of an email address in the string.
- **EMAIL_USERID.** Is the user ID portion of an email address in the string.
- **URL_PROTOCOL.** Is the URL protocol for a URL in the string.
- **URL_HOST.** Is the host name of the URL in the string.
- **URL_PORT.** Is the port number of the URL in the string.
- **URL_PATH.** Is the URL path for a URL in the string.
- **NAME_FIRST.** Is the first token (group of characters) in the string. Tokens are delimited by blanks.

- **NAME_LAST.** Is the last token (group of characters) in the string. Tokens are delimited by blanks.

string

Alphanumeric

Is the string from which the element will be extracted.

Example: Extracting an Element From a String

SPLIT extracts the URL protocol from the string STRING1.

```
SPLIT(URL_PROTOCOL, STRING1)
```

For the URL *'http://www.informationbuilders.com'* in STRING1, the result is *http*.

SUBSTR: Extracting a Substring From a String Value (SQL)

The SUBSTR function returns a substring of a character value. You specify the start position of the substring within the value. You can also specify the length of the substring (if omitted, the substring extends from the start position to the end of the string value). If the specified length value is longer than the input string, the result is the full input string.

SUBSTRING is identical to SUBSTR.

Syntax: How to Extract a Substring From a String Value

```
SUBSTR(arg FROM start-pos [FOR length])
```

or

```
SUBSTR(arg, start-pos [, length])
```

where:

arg

character string

Is the field containing the parent character string.

start-pos

Integer

Is the position within *arg* at which the substring begins.

length

Integer

If present, is the length of the substring. This function returns a varying character string. The data type of the result has a length equal to that of the input argument (although the value may be shorter).

Example: Extracting a Substring From a String Value

SUBSTR function returns a substring. This example,

```
SUBSTR('ABC' FROM 2)
```

Returns BC.

This example,

```
SUBSTRING('ABC' FROM 1 FOR 2)
```

returns AB.

This example,

```
SUBSTR('ABC', 10)
```

returns ABC.

TOKEN: Extracting a Token From a String

The token function extracts a token (substring) from a string of characters. The tokens are separated by a delimiter consisting of one or more characters and specified by a token number reflecting the position of the token in the string.

Syntax: How to Extract a Token From a String

```
TOKEN(string, delimiter, number)
```

where:

string

Fixed length alphanumeric

Is the character string from which to extract the token.

delimiter

Fixed length alphanumeric

Is a delimiter consisting of one or more characters.

TOKEN can be optimized if the delimiter consists of a single character.

number

Integer

Is the token number to extract.

Example: **Extracting a Token From a String**

TOKEN extracts the second token from the PRODUCT_SUBCATEG column, where the delimiter is a blank:

```
TOKEN(PRODUCT_SUBCATEG,' ',2)
```

For *iPod Docking Station*, the result is *Docking*.

TRIM: Removing Leading or Trailing Characters (SQL)

The TRIM function removes leading and/or trailing characters from a character string. The character to be removed may be specified. If no character is specified, the space character is assumed. Whether to remove leading and/or trailing characters may be specified. Without this specification, both leading and trailing appearances of the specified character are removed.

Syntax: **How to Remove Leading or Trailing Characters**

```
TRIM(arg)
TRIM(trim-where [trim-char] FROM arg)
TRIM(trim-char FROM arg)
```

where:

arg

character string

Is the source string value to be trimmed.

trim-where

Value may be LEADING, TRAILING or BOTH. Indicates where characters will be removed. If not specified, BOTH is assumed.

trim-char

character string

Is the character to be removed. If not specified, the space character is assumed.

This function returns a varying character string. The data type of the result has a length equal to that of the input argument (although the value may be shorter).

Example: Removing Leading or Trailing Characters

TRIM removes leading and/or trailing characters. This example,

```
TRIM('  ABC  ')
```

returns ABC.

This example,

```
TRIM(LEADING FROM '  ABC  ')
```

returns 'ABC '.

This example,

```
TRIM(TRAILING FROM '  ABC  ')
TRIM(BOTH 'X' FROM 'XXYYYXXX') = ('YYY')
```

returns ' ABC'

This example,

```
TRIM(BOTH 'X' FROM 'XXYYYXXX')
```

returns YYY.

UCASE: Converting a Character String to Uppercase

The UCASE function converts a character string value to uppercase. That is, lowercase letters are replaced by their corresponding uppercase values. UPPER and UPPERCASE are identical to UCASE.

Syntax: How to Convert a Character String to Uppercase

```
UCASE(arg)
```

where:

arg

character string

Is the value to be converted to uppercase.

This function returns a character string whose length is the same as that of the input argument.

Example: Converting a Character String to Uppercase

UCASE converts a character string value to uppercase. This example,

```
UCASE('abc')
```

returns ABC.

Chapter 20

SQL Date and Time Functions

SQL date and time functions perform manipulations on date and time values.

In this chapter:

CURRENT_DATE: Obtaining the Date

The CURRENT_DATE function returns the current date of the operating system in the form YYYYMMDD.

Syntax: How to Obtain the Current Date

```
CURRENT_DATE
```

This function returns the date in YYMD format.

Example: Obtaining the Current Date

On August 18, 2005, CURRENT_DATE will return 20050818.

CURRENT_TIME: Obtaining the Time

The CURRENT_TIME function returns the current time of the operating system in the form HHMMSS. You may specify the number of decimal places for fractions of a second–0, 3, or 6 places. Zero (0) places is the default.

Syntax: How to Obtain the Current Time

```
CURRENT_TIME[(precision)]
```

where:

precision

Integer constant

Is the number of decimal places for fractions of a second. Possible values are 0, 3, and 6.

This function returns the time (format: HHIS if no decimal places; HHISs if 3 decimal places; HHISsm if 6 decimal places).

Example: Obtaining the Current Time

At exactly half past 11 AM:

CURRENT_TIME returns 113000.

CURRENT_TIME(3) returns 113000000.

CURRENT_TIME(6) returns 113000000000.

CURRENT_TIMESTAMP: Obtaining the Timestamp (Date/Time)

The CURRENT_TIMESTAMP function returns the current timestamp of the operating system (date and time) in the form YYYYMMDDHHMMSS. You may specify the number of decimal places for fractions of a second–0, 3, or 6 places. Six (6) places is the default.

Syntax: How to Obtain the Current Timestamp

```
CURRENT_TIMESTAMP[(precision)]
```

where:

precision

Integer constant

Is the number of decimal places for fractions of a second. Possible values are 0, 3, and 6.

This function returns a timestamp (format: HYYMDS if no decimal places; HYYMDs if 3 decimal places; HYYMDm if 6 decimal places).

Example: Obtaining the Current Timestamp

At 2:11:23 PM on October 9, 2005:

CURRENT_TIMESTAMP returns 20051009141123000000.

CURRENT_TIMESTAMP(0) returns 20051009141123.

CURRENT_TIMESTAMP(3) returns 20051009141123000.

CURRENT_TIMESTAMP(6) returns 20051009141123000000.

CURRENT_TIMEZONE: Obtaining the Time Zone

The CURRENT_TIMEZONE function returns the current time zone of the server in the form HHMMSS.

This function is still under development.

Syntax: How to Obtain the Current Time Zone

```
CURRENT_TIMEZONE()
```

Example: Obtaining the Current Time Zone

In New York City (UTC -5:00) CURRENT_TIMEZONE() returns 50000.

DAY: Obtaining the Day of the Month From a Date/Timestamp

The DAY function returns the day of the month from a date or timestamp value.

Syntax: How to Obtain the Day of the Month From a Date or Timestamp

```
DAY(arg)
```

where:

arg

Date or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Day of the Month From a Date or Timestamp

DAY returns the day of the month from a date or timestamp. This example,

```
DAY('1976-07-04')
```

returns 4.

This example,

```
DAY('2001-01-22 10:00:00')
```

returns 22.

DAYNAME: Returning the Name of the Day From a Date Expression

DAYNAME returns a character string that contains the data-source-specific name of the day for the day part of a date expression.

Syntax: How to Return the Name of the Day From a Date Expression

```
DAYNAME(date_exp)
```

where:

date_exp

Is a date or date-time expression.

Example: Returning the Name of the Day From a Date Expression

DAYNAME returns the name of the day.

```
DAYNAME(TIME_DATE)
```

For January 1, 2009, the result is Thursday.

DAYS: Obtaining the Number of Days Since January 1, 0001

The DAYS function returns 1 more than the number of days from January 1, 0001 to the provided date value.

Syntax: How to Obtain the Number of Days Since January 1, 1900

```
DAYS(arg)
```

where:

arg

Date or timestamp

Is the input argument.

This function returns an integer value.

Example: Obtaining the Number of Days Since January 1, 1900

DAYS returns one more than the number of days since January 1, 1900.

```
DAYS('2000-01-01')
```

returns 730120.

DAY_OF_YEAR: Returning the Numeric Day of the Year

DAY_OF_YEAR takes a date or date-time argument and returns the number of the day within the year that represents that date.

Syntax: How to Return the Numeric Day of the Year

```
DAY_OF_YEAR(date)
```

where:

date

Is a date or date-time field or a date or date-time value enclosed in single quotation marks.

Example: **Returning the Numeric Day of the Year**

DAY_OF_YEAR('1976-07-04') returns 186.

DAY_OF_YEAR('2012-02-24 10:00:00') returns 55.

DTDIFF: Returning the Number of Component Boundaries Between Date or Date-Time Values

Given two dates in standard date or date-time formats, DTIFF returns the number of given component boundaries between the two dates. The returned value has integer format for calendar components or double precision floating point format for time components.

Syntax: **How to Return the Number of Component Boundaries**

```
DTDIFF(end_date, start_date, component)
```

where:

end_date

Date or date-time

Is the ending full-component date in either standard date or date-time format. If this date is given in standard date format, all time components are assumed to be zero.

start_date

Date or date-time

Is the starting full-component date in either standard date or date-time format. If this date is given in standard date format, all time components are assumed to be zero.

component

Keyword

Is the component on which the number of boundaries is to be calculated. For example, QUARTER finds the difference in quarters between two dates. Valid components (and acceptable values) are:

- YEAR (1-9999).
- QUARTER (1-4).
- MONTH (1-12).
- WEEK (1-53). This is affected by the WEEKFIRST setting.
- DAY (of the Month, 1-31).

- ❑ HOUR (0-23).
- ❑ MINUTE (0-59).
- ❑ SECOND (0-59).

Example: Returning the Number of Years Between Two Dates

DTDIFF calculates employee age when hired:

```
DTDIFF(START_DATE, DATE_OF_BIRTH, YEAR)
```

For the date of birth 1991/06/04 and the start date 2008/11/14, the result is 17.

DTDIFF calculates the difference between two date-time values in minutes:

```
DTDIFF(DATETIME1, DATETIME2, MINUTES)
```

For DATETIME1 = 2020/0116 12:25 and DATETIME2 = 2020/0116 12:20, the result is 5.

For DATETIME1 = 2020/0116 12:25 and DATETIME2 = 2020/0115 12:20, the result is 1445.

DTRUNC: Returning the Start of a Date Period for a Given Date

Given a date or timestamp and a component, DTRUNC returns the first date within the period specified by that component.

Syntax: How to Return the First or Last Date of a Date Period

```
DTRUNC(date_or_timestamp, date_period)
```

where:

date_or_timestamp
Date or date-time

Is the date or timestamp of interest, which must provide a full component date.

date_period
Is the period whose starting or ending date you want to find. Can be one of the following:

- ❑ DAY, returns the date that represents the input date (truncates the time portion, if there is one).
- ❑ YEAR, returns the date of the first day of the year.
- ❑ MONTH, returns the date of the first day of the month.

- QUARTER, returns the date of the first day in the quarter.
- WEEK, returns the date that represents the first date of the given week.

  By default, the first day of the week will be Sunday, but this can be changed using the WEEKFIRST parameter.
- YEAR_END, returns the last date of the year.
- QUARTER_END, returns the last date of the quarter.
- MONTH_END, returns the last date of the month.
- WEEK_END, returns the last date of the week.

Example: Returning the First Date in a Date Period

DTRUNC returns the first date of the quarter given the date of birth:

```
DTRUNC(DATE_OF_BIRTH,QUARTER)
```

For 1993/03/27, the result is 1993/03/01.

Example: Using the Start of Week Parameter for DTRUNC

DTRUNC returns the date that represents the start of the week.

```
DTRUNC(START_DATE, WEEK)
```

For 2013/01/15, the result is 2013/01/13

Example: Returning the Date of the Last Day of a Week

DTRUNC calculates the date of the end of the week.

```
WEEKEND/YYMD = DTRUNC(START_DATE, WEEK_END)
```

For 2013/01/15, the result is 2013/01/19.

INTERVAL: Adding an Interval to a Date or Date-Time Value

INTERVAL adds a time interval to a date or date-time value. All date and time components are acceptable intervals.

Syntax: How to Add an Interval to a Date or Date-Time Value

```
datetime + INTERVAL increment component
```

where:

datetime

Is a date or date-time field, or a date or date-time value enclosed in single quotation marks.

increment

Is the number of units of the date component to add.

component

Is the date or time component to which to add the increment. Supported components are:

- **Components for date fields.** YEAR, QUARTER, MONTH, WEEK, DAY.
- **Components for date-time fields.** YEAR, QUARTER, MONTH, WEEK, DAY, HOUR, MINUTE, SECOND, MILLISECOND, MICROSECOND.
- **Components for time fields.** HOUR, MINUTE, SECOND, MILLISECOND, MICROSECOND.

Example: Adding an Interval to a Date or Date-Time Value

'2001/03/24' + INTERVAL 2 MONTH returns 2001/05/24.

'2005/02/15' + INTERVAL 3 YEAR returns 2008/02/15.

'2005/02/15 12:33:11' + INTERVAL 15 MINUTE returns 2005/02/15 12:48:11

EXTRACT: Obtaining a Datetime Field From Date/Time/Timestamp

The EXTRACT function can be used to obtain the year, month, day of month, hour, minute, second, millisecond, or microsecond component of a date, time, or timestamp value.

Syntax: How to Obtain a Datetime Field From a Date, Time, or Timestamp

```
EXTRACT(field FROM arg)
```

where:

arg

Date, time, or timestamp

Is the input argument.

field

Is the datetime field of interest. Possible values are YEAR, QUARTER, MONTH, DAY, WEEKDAY, HOUR, MINUTE, SECOND, MILLISECOND and MICROSECOND.

This function returns an integer value.

Note:

- YEAR, QUARTER, MONTH, DAY, and WEEKDAY can be used only if the argument is date or timestamp.
- HOUR, MINUTE, SECOND, MILLISECOND and MICROSECOND can be used only if the argument is time or timestamp.

Example: Obtaining a Datetime Field From a Date, Time, or Timestamp

EXTRACT returns the components of a date, time, or timestamp. This example,

```
EXTRACT(YEAR FROM '2000-01-01')
```

returns 2000.

This example,

```
EXTRACT(HOUR FROM '11:22:33')
```

returns 11.

This example,

```
EXTRACT(MICROSECOND FROM '2000-01-01 11:22:33.456789')
```

returns 456,789.

HOUR: Obtaining the Hour From Time/Timestamp

The HOUR function returns the hour field from a time or timestamp value.

Syntax: How to Obtain the Hour From a Time or Timestamp

```
HOUR(arg)
```

where:

arg

Time or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Hour From a Time or Timestamp

HOUR returns the hour from a time or timestamp. This example,

```
HOUR('11:22:33')
```

returns 11.

This example,

```
HOUR('2001-01-22 10:00:00')
```

returns 10.

MICROSECOND: Obtaining Microseconds From Time/Timestamp

The MICROSECOND function returns the number of microseconds from a time or timestamp value.

Syntax: How to Obtain the Number of Microseconds From a Time or Timestamp

```
MICROSECOND(arg)
```

where:

`arg`

Time or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Number of Microseconds From a Time or Timestamp

MICROSECOND returns the microseconds from a time or timestamp. This example,

```
MICROSECOND('11:22:33.456789')
```

returns 456,789.

This example,

```
MICROSECOND('2001-01-22 10:00:00')
```

returns 0.

MILLISECOND: Obtaining Milliseconds From Time/Timestamp

The MILLISECOND function returns the number of milliseconds from a time or timestamp value.

Syntax: How to Obtain the Number of Milliseconds From a Time or Timestamp

```
MILLISECOND(arg)
```

where:

arg

Time or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Number of Milliseconds From a Time or Timestamp

MILLISECOND returns the number of milliseconds from a time or timestamp. This example,

```
MILLISECOND('11:22:33.456')
```

returns 456.

This example,

```
MILLISECOND('2001-01-22 10:11:12')
```

returns 0.

MINUTE: Obtaining the Minute From Time/Timestamp

The MINUTE function returns the number of minutes from a time or timestamp value.

Syntax: How to Obtain the Minute From a Time or Timestamp

```
MINUTE(arg)
```

where:

arg

Time or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Minute From a Time or Timestamp

MINUTE returns the minutes from a time or timestamp. This example,

```
MINUTE('11:22:33')
```

returns 22.

This example,

```
MINUTE('2001-01-22 10:11:12')
```

returns 11.

MONTH: Obtaining the Month From Date/Timestamp

The MONTH function returns the month field from a date or timestamp value.

Syntax: How to Obtain the Month From a Date or Timestamp

```
MONTH(arg)
```

where:

arg

Date or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Month From a Date or Timestamp

MONTH returns the month from a date or timestamp. This example,

```
MONTH('1976-07-04')
```

returns 7.

This example,

```
MONTH('2001-01-22 10:00:00')
```

returns 1.

MONTHNAME: Returning the Name of the Month From a Date Expression

MONTHNAME returns a character string that contains the data-source-specific name of the month for the month part of a date expression.

Syntax: How to Return the Name of the Month From a Date Expression

```
MONTHNAME(date_exp)
```

where:

date_exp
: Is a date or date-time expression.

Example: Returning the Name of the Month From a Date Expression

MONTHNAME returns the name of the month.

```
MONTHNAME(DATE)
```

For 'August 3, 2020', the result is August.

QUARTER: Returning the Quarter of the Year

Given a date or date-time value, QUARTER returns an integer (from 1 to 4) that represents the quarter within which that date falls.

Syntax: How to Return the Quarter of the Year

```
QUARTER(arg)
```

where:

arg
: Date or date-time

Is the input date or date-time value.

Example: Returning the Quarter of the Year

QUARTER returns the quarter of the year for each date of birth:

```
QUARTER(DATE_OF_BIRTH)
```

For 1993/03/27, the result is 1.

SECOND: Obtaining the Second Field From Time/Timestamp

The SECOND function returns the second field from a time or timestamp value.

Syntax: How to Obtain the Second Field From a Time or Timestamp

```
SECOND(arg)
```

where:

arg

Time or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Second Field From a Time or Timestamp

SECOND returns seconds from a time or timestamp. This example,

```
SECOND('11:22:33')
```

returns 33.

This example,

```
SECOND('2001-01-22 12:24:36')
```

returns 36.

WEEKDAY: Returning the Day of the Week

Given a date or date-time value, WEEKDAY returns an integer from 1 (Monday) to 7 (Sunday) representing the day of the week for that date.

Syntax: How to Return the Day of the Week

```
WEEKDAY(arg)
```

where:

arg

Date or date-time

Is the input date or date-time value.

Example: Returning the Day of the Week

WEEKDAY returns the day of the week for each birth date, where 1 represents Monday and 7 represents Sunday:

```
WEEKDAY(DATE_OF_BIRTH)
```

For 1993/03/27, the result is 6 (Saturday).

YEAR: Obtaining the Year From a Date or Timestamp

The YEAR function returns the year field from a date or timestamp value.

Syntax: How to Obtain the Year From a Date or Timestamp

```
YEAR(arg)
```

where:

arg

Date or timestamp

Is the input value.

This function returns an integer value.

Example: Obtaining the Year From a Date or Timestamp

YEAR returns the year from a date or timestamp value. This example,

```
YEAR('1976-07-04')
```

returns 1976.

This example,

```
YEAR('2001-01-22 10:00:00')
```

returns 2001.

Chapter 21

SQL Data Type Conversion Functions

SQL data type conversion functions convert fields from one data type to another.

In this chapter:

CAST: Converting to a Specific Data Type

The CAST function converts the value of its argument to a specified data type.

Syntax: **How to Convert to a Specific Data Type**

```
CAST(expression AS data_type[(length)])
```

where:

arg

Any data type that can be converted to the result data type

Is the value to be converted.

data-type

Is the result data type: CHARACTER, CHARACTER VARYING, NUMERIC, DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, DOUBLE PRECISION, DATE, TIME or TIMESTAMP.

length

Is an optional parameter of character data types.

This function returns the input value converted to the specified data type.

Example: Converting to a Specific Data Type

CAST converts a value to a specified data type. This example,

```
CAST(2.5 AS INTEGER)
```

returns 2.

This example,

```
CAST('3.333' AS FLOAT)
```

returns 3.333.

CHAR: Converting to a Character String

There are two versions of the CHAR function, one for converting an argument to a character string, and one for converting a date, time, or timestamp value to a standard format. The version that takes one argument converts its argument to a character string. For information about using CHAR to convert a date, time, or timestamp value to a standard format, see *CHAR: Converting to a Standard Date-Time Format* on page 439.

Syntax: How to Convert to a Character String

```
CHAR(arg)
```

where:

arg

Any type

Is the value to be converted.

This function returns a character string whose length is of sufficient size to hold the value.

Example: Converting to a Character String

CHAR converts a value to a character string. This example,

```
CHAR(566.23)
```

returns 566.23.

CHAR: Converting to a Standard Date-Time Format

There are two versions of the CHAR function, one for converting an argument to a character string, and one for converting a date, time, or timestamp value to a standard format. The version that takes two arguments converts a date, time, or timestamp value to one of the standard date-time formats. For information about using CHAR to convert a single argument to a character string, see *CHAR: Converting to a Character String* on page 438.

Syntax: How to Convert a Date, Time, or Timestamp Value to a Standard Format

```
CHAR(datetime, fmt)
```

where:

datetime

Date

Is the date, time, or timestamp value to be converted.

fmt

Can be one of the following formats:

Name of Standard	Date Format	Time Format	Timestamp Format
ISO	yyyy-mm-dd	hh.mm.ss	yyyy-mm-dd hh:mm:ss.xxxxxx
USA	mm/dd/yyyy	hh.mm AM/PM	yyyy-mm-dd-hh.mm.ss.xxxxxx
EUR	dd.mm.yyyy	hh.mm.ss	yyyy-mm-dd-hh.mm.ss.xxxxxx
JIS	yyyy-mm-dd	hh:mm:ss	yyyy-mm-dd-hh.mm.ss.xxxxxx

This function returns a character string whose length is of sufficient size to hold the value.

Example: **Converting Date and Time Values to Standard Formats**

CHAR converts a date, time, or timestamp value to a standard format. The following examples use the constants CURRENT DATE, CURRENT TIME, and CURRENT TIMESTAMP. Assume the current date is November 17, 2011:

`CHAR(CURRENT DATE, USA)` returns `11/17/2011`

`CHAR(CURRENT DATE, ISO)` returns `2011-11-17`

`CHAR (CURRENT TIME, USA)` returns `03:45 PM`

`CHAR (CURRENT TIME, ISO)` returns `15.45.00`

`CHAR(CURRENT TIMESTAMP, ISO)` returns `2011-11-17 15:45:00`

DATE: Converting to a Date

The DATE function converts its argument to a date. The type of the argument value may be character, date, or timestamp.

If the argument is:

- ❏ A character, its value must correctly represent a date; that date is the result.
- ❏ A date, its value is returned.
- ❏ A timestamp, the date portion of the timestamp value is returned.

Syntax: **How to Convert to a Date**

```
DATE(arg)
```

where:

`arg`

  character string, date, or timestamp

  Is the value to be converted.

The DATE function returns a date in YYMD format.

Example: **Converting to a Date**

DATE converts a value to a date. This example,

```
DATE('1999-03-29 14:39:30')
```

returns 19990329.

DECIMAL: Converting to Decimal Format

The DECIMAL function converts a number to fixed-length decimal format.

Syntax: How to Convert to the Decimal Format

```
DECIMAL(arg, [length [,dec-places]])
```

where:

arg

Numeric

Is the input value.

length

Integer

The maximum number of digits in the integer portion of the result. The default is 15.

dec-places

Integer

Is the number of decimal places in the result. The default is the same number of decimal places as in the type of the argument.

This function returns a numeric value in fixed-length decimal format.

Example: Converting to Decimal Format

DECIMAL converts a number to fixed-length decimal format. This example,

```
DECIMAL(5.12345, 4, 2)
```

returns 5.12.

DIGITS: Converting a Numeric Value to a Character String

The DIGITS function extracts the digits of a decimal or integer value into a character string. The sign and decimal point of the number (if present) are ignored.

Syntax: How to Convert a Numeric Value to a Character String

```
DIGITS(arg)
```

where:

arg

Numeric (decimal or integer, not floating-point)

Is the numeric value.

The length of the resulting string is determined by the precision of the argument.

Example: Converting a Numeric Value to a Character String

DIGITS converts a numeric value to a character string. This example,

```
DIGITS(-444.321)
```

returns 0000444321.

DT_FORMAT: Converting a Date or Date-Time Value to an Alphanumeric String

DT_FORMAT converts a date or date-time value to an alphanumeric string in a specified date or date-time format. For information about date and date-time formats, see the *Describing Data With WebFOCUS Language* manual.

Syntax: How to Convert a Date Value to an Alphanumeric String in a Specified Date Format

```
DT_FORMAT(date,'date_format')
```

where:

date

Numeric, date, or date-time

Is the date or date-time field or value to be converted.

'*date_format*'

Alphanumeric literal

Is a date or date-time format that fits the input date format type, enclosed in single quotation marks.

Example: Converting Date and Date_Time Values to Alphanumeric Format

DT_FORMAT converts the current date and time down to the seconds to a string in date-time format HYYMTDs:

```
DT_FORMAT( DT_CURRENT_DATETIME(SECOND),'HYYMTDs')
```

On December 17, 2019 at approximately 11:36 A.M., the result is:

```
2019 December 17 11:36:45.000
```

FLOAT: Converting to Floating Point Format

The FLOAT function converts a number to floating-point format.

Syntax: How to Convert to the Floating Point Format

```
FLOAT(arg)
```

where:

`arg`

Numeric

Is the input value.

This function returns the value in floating-point format.

Example: Converting to Floating Point Format

FLOAT converts a number to floating-point format. This example,

```
FLOAT(3)
```

returns 3.0.

FOCDATE: Converting any Date Value to YYMD Date Format

FOCDATE takes a date in alphanumeric or integer format with year, month, and day options, or in date format, and returns a value in YYMD format.

Syntax: How to Convert any Date Value to YYMD Date Format

```
FOCDATE(date, 'YYMD')
```

where:

`date`

Is a date field, an alphanumeric or integer field with date display options, or a date literal enclosed in single quotation marks. The date can be a valid combination of year, month, and/or day options. If the day is not specified, it defaults to 01. Only four-digit years are supported in a date literal.

`'YYMD'`

Is the format of the result returned. You can specify any date format that has only date components. However the result is always a date in YYMD format.

For FOCDATE('November 23, 2010' , 'MDYY'), the result is 2010/11/23.

For FOCDATE('11/23/2010' , 'DMYY'), the result is 2010/11/23.

INT: Converting to an Integer

The INT function converts a number to an integer. If the input value is not an integer, the result is truncated.

INTEGER is identical to INT.

Syntax: How to Convert to an Integer

```
INT(arg)
```

where:

`arg`

Numeric

Is the input value.

This function returns the number in integer format.

Example: Converting to an Integer

INT converts a number to an integer. This example,

```
INT(4.8)
```

returns 4.

OLDDATE: Converting Any Date Value to Alphanumeric Format With Date Options

OLDDATE takes a date in alphanumeric or integer format with year, month and day options, or in date format, and returns a value in alphanumeric format with year, month, and/or day options.

Syntax: How to Convert any Date Value to YYMD Date Format

```
OLDDATE(date, 'format')
```

where:

date

Is a date field, an alphanumeric or integer field with date display options, or a date literal enclosed in single quotation marks. The date can be a valid combination of year, month, and/or day options. If the day is not specified, it defaults to 01. Only four-digit years are supported in a date literal.

format

Is the format of the result returned, enclosed in single quotation marks. You can specify any date format that has only date components. The result will be a date in alphanumeric format with the specified date options in the specified order, such as 'A8YYMD'.

For FOCDATE('November 23, 2010' , 'MDYY'), the result is '11/23/2010'.

For FOCDATE('11/23/2010' , 'YMD'), the result is '10/11/23'.

PHONETIC: Returning a Phonetic Key for a String

PHONETIC calculates a phonetic key for a string, or a null value on failure. Phonetic keys are useful for grouping alphanumeric values, such as names, that may have spelling variations. This is done by generating an index number that will be the same for the variations of the same name based on pronunciation. One of two phonetic algorithms can be used for indexing, Metaphone and Soundex. Metaphone is the default algorithm, except on z/OS where the default is Soundex.

You can set the algorithm to use with the following command.

```
SET PHONETIC_ALGORITHM = {METAPHONE|SOUNDEX}
```

Most phonetic algorithms were developed for use with the English language. Therefore, applying the rules to words in other languages may not give a meaningful result.

Metaphone is suitable for use with most English words, not just names. Metaphone algorithms are the basis for many popular spell checkers.

Note: Metaphone is not optimized in generated SQL. Therefore, if you need to optimize the request for an SQL DBMS, the SOUNDEX setting should be used.

Soundex is a legacy phonetic algorithm for indexing names by sound, as pronounced in English.

Syntax: How to Return a Phonetic Key

```
PHONETIC(string)
```

where:

`string`

Alphanumeric

Is a string for which to create the key. A null value will be returned on failure.

Example: Generating a Phonetic Key

PHONETIC generates a phonetic key for LAST_NAME:

```
PHONETIC(LAST_NAME)
```

For last names SMITH and SMYTHE, the same phonetic key, S530, is generated.

SMALLINT: Converting to a Small Integer

The SMALLINT function converts a number to a small integer. Generally, a small integer occupies only two bytes in memory.

Syntax: How to Convert to a Small Integer

```
SMALLINT(arg)
```

where:

`arg`

Numeric

Is the input value.

This function returns the number in small integer format.

Example: Converting to a Small Integer

SMALLINT converts a number to a small integer. This example,

```
SMALLINT(3.5)
```

returns 3.

TIME: Converting to a Time

The TIME function converts its argument to a time. The type of the argument value may be character, time, or timestamp.

- ❑ If the argument is a character, its value must correctly represent a time; that time is the result.
- ❑ If the argument is a time, its value is returned.
- ❑ If the argument is a timestamp, the time portion of the timestamp value is returned.

Syntax: How to Convert to a Time

```
TIME(arg)
```

where:

arg

character string, time, or timestamp

Is the input value.

This function returns a time.

Example: Converting to a Time

TIME converts a value argument to a time. This example,

```
TIME('2004-03-15 01:02:03.444')
```

returns 010203444.

TIMESTAMP: Converting to a Timestamp

The TIMESTAMP function converts its argument to a timestamp. The argument type can be character, date, time, or timestamp.

- ❑ If the argument is a character, its value must correctly represent a timestamp; that timestamp is the result.
- ❑ If the argument is a date, the value of the result is the timestamp, with the date component equal to the argument and the time component equal to midnight.
- ❑ If the argument is a time, the value of the result is the timestamp, with the date component equal to the current date, and the time component equal to the argument.

- ❏ If the argument is a timestamp, its value is returned.

Syntax: How to Convert to a Timestamp

```
TIMESTAMP(arg)
```

where:

`arg`

character string, date, time, or timestamp

Is the input value.

This function returns a timestamp.

Example: Converting to a Timestamp

TIMESTAMP converts a value to a timestamp. This example,

```
TIMESTAMP('2004-06-24')
```

returns 20040624000000.

This example,

```
TIMESTAMP('11:22:33')
```

returns 20010101112233, if the current date is January 1, 2001.

VARGRAPHIC: Converting to Double-byte Character Data

The VARGRAPHIC function converts the input value to double-byte character data

Syntax: How to Convert to the Double-byte Character Format

```
VARGRAPHICarg
```

where:

`arg`

character, graphic, or date

Is the input value.

Note: This function can only be used for DB2 and can only be used with Direct or Automatic Passthru. This function returns the value in double-byte character format.

Chapter 22

SQL Numeric Functions

SQL numeric functions perform calculations on numeric constants and fields.

In this chapter:

ABS: Returning an Absolute Value (SQL)

The ABS function returns the absolute value of a number.

Syntax: **How to Return an Absolute Value**

```
ABS(arg)
```

where:

arg

Numeric

Is the input value.

This function returns the value as the same data type as the argument. For example, if the argument is an integer, the result will be also be an integer.

Example: Returning an Absolute Value

ABS returns the absolute value of a number. This example,

```
ABS(-5.5)
```

returns 5.5.

CEIL: Returning the Smallest Integer Greater Than or Equal to a Value

CEIL returns the smallest integer value not less than the argument. CEILING is a synonym for CEIL.

Syntax: How to Return the Smallest Integer Greater Than or Equal to a Value

```
CEIL(n)
```

where:

`n`

Numeric or Alphanumeric

Is the value less than or equal to the returned integer. For exact-value numeric arguments, the return value has an exact-value numeric type. For alphanumeric or floating-point arguments, the return value has a floating-point type.

Example: Returning an Integer Greater Than or Equal to a Value

CEIL returns an integer greater than or equal to the argument.

```
CEIL(N)
```

For N=1.23, the result is 2.

For N=-1.23, the result is -1.

EXP: Returning e Raised to a Power

The EXP function returns the mathematical constant e raised to a power.

Syntax: How to Return e Raised to a Power

```
EXP(arg)
```

where:

`arg`

Numeric

Is the value of the power to which to raise the mathematical constant e.

Example: **Returning e Raised to a Power**

EXP returns the mathematical constant e to a power. This example,

```
EXP(4)
```

returns 54.598.

FLOOR: Returning the Largest Integer Less Than or Equal to a Value (SQL)

FLOOR returns the largest integer value not greater than a value.

Syntax: **How to Return the Largest Integer Less Than or Equal to a Value**

```
FLOOR(n)
```

where:

n

Numeric or Alphanumeric

Is the value greater than or equal to the returned integer. For exact-value numeric arguments, the return value has an exact-value numeric type. For alphanumeric or floating-point arguments, the return value has a floating-point type.

Example: **Returning an Integer Less Than or Equal to a Value**

FLOOR returns an integer less than or equal to the argument.

```
FLOOR(N)
```

For N=1.23, the result is 1.

For N=-1.23, the result is -2.

LOG: Returning a Logarithm (SQL)

The LOG function returns the natural logarithm of the input value.

Syntax: **How to Return a Logarithm**

```
LOG(arg)
```

where:

arg

Numeric

Is the input value.

This function returns double precision numbers with three decimal places.

Example: Returning a Logarithm

LOG returns the natural logarithm of a value. This example,

```
LOG(4)
```

returns 1.386.

LOG10: Calculating the Base 10 Logarithm

LOG10 returns the base-10 logarithm of a numeric expression.

Syntax: How to Calculate the Base 10 Logarithm

```
LOG10(num_exp)
```

where:

num_exp

Numeric

Is the numeric value for which to calculate the base 10 logarithm.

Example: Calculating the Base 10 Logarithm

LOG10 calculates the base 10 log of NUMBER.

```
LOG10(NUMBER)
```

For 145, the result is 2.161.

MOD: Returning the Remainder of a Division

The SQL function MOD returns the remainder of the first argument divided by the second argument.

Syntax: How to Return the Remainder of a Division

```
MOD(n,m)
```

where:

n

Numeric

Is the dividend (number to be divided).

m

Numeric

Is the divisor (number to divide by). If the divisor is zero (0), MOD returns NULL.

Example: Returning the Remainder of a Division

MOD returns the remainder of *n* divided by *m*.

```
MOD(N,M)
```

For N=16 and M=5, the result is 1.

For N=34.5 and M=3, the result is 1.5.

POWER: Raising a Value to a Power (SQL)

The POWER function returns the value calculated by raising the first argument to the power specified by the second argument.

Syntax: How to Return a Value Raised to a Power

```
POWER(arg1, arg2)
```

where:

arg1

Numeric

Is the value to be raised to the power specified by *arg2*.

arg2

Numeric

Is the value of the power to which to raise *arg1*.

Example: Returning a Value Raised to a Power

POWER returns the value calculated by raising the first argument to the value specified by the second argument. This example,

```
EXP(2,4)
```

returns 16.000.

RAND: Producing a Stream of Random Numbers

RAND produces a stream of random numbers uniformly distributed between zero and 1. The stream of numbers is reproducible. If you provide the same seed in multiple runs, you will get the same stream of numbers.

Syntax: How to Produce a Stream of Random Numbers

```
RAND([seed])
```

where:

seed

Is a number or a field containing the number that is to be used as the starting point for the random number generation. If ommitted, a default seed will be used.

Example: Producing a Stream of Random Numbers

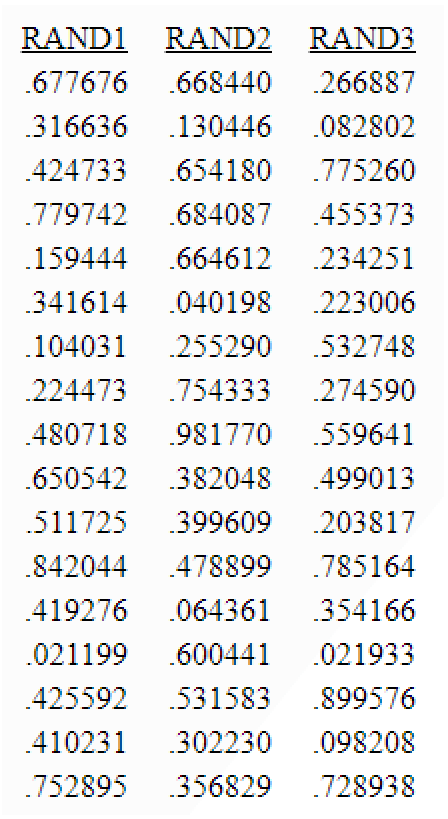
The following request uses the dminv table created by running the *DataMigrator - General* tutorial. It generates three random number streams:

- ❏ RAND1 uses the same field value as the seed and, therefore, produces a reproducible stream of numbers.
- ❏ RAND2 uses the time of day as the seed and, therefore, produces a different stream of numbers as the time of day changes.
- ❏ RAND3 uses the default seed and, therefore, produces a reproducible stream of numbers.

```
SQL
SELECT
 RAND(T1.QTY_IN_STOCK) AS RAND1,
 RAND(CAST(EDIT('&TOD','99$99$99') AS INTEGER))  AS RAND2,
 RAND() AS RAND3
FROM
DMINV T1;

TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

Partial output is shown in the following image.



```
RAND1    RAND2    RAND3
.677676  .668440  .266887
.316636  .130446  .082802
.424733  .654180  .775260
.779742  .684087  .455373
.159444  .664612  .234251
.341614  .040198  .223006
.104031  .255290  .532748
.224473  .754333  .274590
.480718  .981770  .559641
.650542  .382048  .499013
.511725  .399609  .203817
.842044  .478899  .785164
.419276  .064361  .354166
.021199  .600441  .021933
.425592  .531583  .899576
.410231  .302230  .098208
.752895  .356829  .728938
```

ROUND: Rounding a Number to a Given Number of Decimal Places

Given a numeric expression and an integer count, ROUND returns the numeric expression rounded to that number of decimal places. If the number of decimal places is negative, it rounds to the left of the decimal point.

Syntax: How to Round a Number to a Given Number of Decimal Places

```
ROUND(num_exp, count)
```

where:

num_exp

Numeric

Is the numeric expression to be rounded.

count

Numeric

Is the number of decimal places to which the numeric expression is to be rounded. If the number of decimal places is negative, ROUND rounds to the left of the decimal point.

Example: **Rounding a Number to a Given Number of Decimal Places**

ROUND rounds the number 1234.56 to -3 decimal places.

```
ROUND(1.23456, 3)
```

The result is 1.23500.

ROUND rounds the number 1.23456 to 3 decimal places.

```
ROUND(1234.56, -3)
```

The result is 1000.00.

SIGN: Returning the Sign of a Number

SIGN takes a numeric argument and returns the value -1 if the number is negative, 0 (zero) if the number is zero, and 1 if the number is positive.

Syntax: **How to Return the Sign of a Number**

```
SIGN(number)
```

where:

number

Is a field containing a numeric value or a number.

Example: **Returning the Sign of a Number**

SIGN(-5.5) returns -1.

SIGN(4) returns 1.

SIGN(0) returns 0.

SQRT: Returning a Square Root (SQL)

The SQRT function returns the square root of the input value.

Syntax: **How to Return a Square Root**

```
sqrt(arg)
```

where:

arg

Numeric

Is the input value.

This function returns double precision numbers with three decimal places.

Example: Returning a Square Root

SQRT returns the square root of a value. This example,

```
SQRT(4)
```

returns 2.000.

TRUNCATE: Truncating a Number to a Given Number of Decimal Places

Given a numeric expression and an integer count, TRUNCATE returns the numeric expression truncated to that number of decimal places. If the number of decimal places is negative, it truncates to the left of the decimal point.

Syntax: How to Truncate a Number to a Given Number of Decimal Places

```
TRUNCATE(num_exp, count)
```

where:

num_exp

Numeric

Is the numeric expression to be truncated.

count

Numeric

Is the number of decimal places to which the numeric expression is to be truncated. If the number of decimal places is negative, TRUNCATE truncates to the left of the decimal point.

Example: Truncating a Number to a Given Number of Decimal Places

TRUNCATE truncates 1.23456 to 3 decimal places.

```
TRUNCATE(1.23456, 3)
```

The result is 1.23400.

Chapter 23

SQL Miscellaneous Functions

The SQL functions described in this chapter perform a variety of conversions, tests, and manipulations.

In this chapter:

ASCII: Returning the ASCII Code for the Leftmost Character in a String

ASCII takes a character string and returns the ASCII code in integer format for the leftmost character in the string.

Syntax: How to Return the ASCII Code for the Leftmost Character in a String

```
ASCII(charexp)
```

where:

charexp

Is any character string.

Example: Returning the ASCII Code for the Leftmost Character in a String

ASCII returns the ASCII code of the leftmost character of CATEGORY.

```
ASCII(CATEGORY)
```

For Coffee, the result is 67.

CHR: Returning the ASCII Character Given a Numeric Code

Given a number code as an argument, CHR returns the ASCII character.

Syntax: How to Return the ASCII Character Given a Numeric Code

```
CHR(number)
```

where:

`number`

Numeric

Is the numeric code to be translated to an ASCII character.

Example: Returning the ASCII Character Given a Numeric Code

CHR translates the numeric code 190 to ASCII.

```
CHR(190)
```

The result is ¾.

COUNTBY: Incrementing Column Values Row by Row

The COUNTBY function produces a column whose values are incremented row by row by a specified amount.

Syntax: How to Increment Column Values Row by Row

```
COUNTBY(arg)
```

where:

`arg`

Integer

Is the value that is incremented for each record.

This function returns an integer value.

Example: Incrementing Column Values Row by Row

In the query,

```
SELECT COUNTBY(1), COUNTBY(2) FROM T
```

the first column takes on the values 1, 2, 3, ..., and the second column takes on the values 2, 4, 6, ...

CURRENT_EDASQLVERSION: Retrieving the SQL Parser Version

CURRENT_EDASQLVERSION returns the date and time of the version of the SQL parser being used.

Syntax: How to Retrieve the SQL Parser Version

```
CURRENT_EDASQLVERSION()
```

Example: Retrieving the SQL Parser Version

CURRENT_EDASQLVERSION sample output is:

```
Dec 20 2019-18:03:23
```

DB_EXPR: Inserting an SQL Expression Into a Request (SQL)

The DB_EXPR function inserts a native SQL expression exactly as entered into the native SQL generated for a FOCUS or SQL language request.

The DB_EXPR function can be used in a DEFINE command, a DEFINE in a Master File, a WHERE clause, a FILTER FILE command, a filter in a Master File, or in an SQL statement. It can be used in a COMPUTE command if the request is an aggregate request (uses the SUM, WRITE, or ADD command) and has a single display command. The expression must return a single value.

Syntax: How to Insert an SQL Expression Into a Request With DB_EXPR

```
DB_EXPR(native_SQL_expression)
```

where:

native_SQL_expression

Is a partial native SQL string that is valid to insert into the SQL generated by the request. The SQL string must have double quotation marks (") around each field reference, unless the function is used in a DEFINE with a WITH phrase.

Reference: Usage Notes for the DB_EXPR Function

- ❏ The expression must return a single value.
- ❏ Any request that includes one or more DB_EXPR functions must be for a synonym that has a relational SUFFIX.
- ❏ Field references in the native SQL expression must be within the current synonym context.
- ❏ The native SQL expression must be coded inline. SQL read from a file is not supported.

Example: Inserting the DB2 BIGINT and CHAR Functions Into a TABLE Request

The following TABLE request against the WF_RETAIL data source uses the DB_EXPR function in the COMPUTE command to call two DB2 functions. It calls the BIGINT function to convert the squared revenue to a BIGINT data type, and then uses the CHAR function to convert that value to alphanumeric.

```
TABLE FILE WF_RETAIL
SUM REVENUE NOPRINT
AND COMPUTE BIGREV/A31 = DB_EXPR(CHAR(BIGINT("REVENUE" * "REVENUE") ) ) ;
AS 'Alpha Square Revenue'
BY REGION
ON TABLE SET PAGE NOPAGE
END
```

The trace shows that the expression from the DB_EXPR function was inserted into the DB2 SELECT statement:

```
  SELECT
  T11."REGION",
   SUM(T1."Revenue"),
   ((CHAR(BIGINT( SUM(T1."Revenue") *  SUM(T1."Revenue")) ) ))
   FROM
  wrd_fact_sales T1,
  wrd_dim_customer T5,
  wrd_dim_geography T11
   WHERE
  (T5."ID_CUSTOMER" = T1."ID_CUSTOMER") AND
  (T11."ID_GEOGRAPHY" = T5."ID_GEOGRAPHY")
   GROUP BY
  T11."REGION  "
   ORDER BY
  T11."REGION  "
   FOR FETCH ONLY;
END
```

GREATEST: Returning the Maximum Value From a List of Arguments

GREATEST returns the maximum value from a list of arguments.

Syntax: **How to Return the Maximum Value From a List of Arguments**

```
GREATEST(arg1, arg2, ...)
```

where:

arg1, arg2, ...

Numeric

Is a list of numeric arguments, which can be fields or literals.

Example: **Returning the Maximum Value From a List of Arguments**

GREATEST returns either the value of the ED_HRS field, or the constant 30, whichever is larger:

```
GREATEST(ED_HRS, 30)
```

For ED_HRS = 45.00, the result is 45.00.

For ED_HRS = 25.00, the result is 30.00.

HEX: Converting to Hexadecimal

The HEX function converts its input value to hexadecimal.

Note: This function is available only for DB2, Ingres, and Informix.

Syntax: **How to Convert to Hexadecimal**

```
HEX(character)
```

where:

character

Is the input value.

This function returns an alphanumeric value.

Example: **Converting a Value to Hex**

This example,

```
HEX('n')
```

returns 6E.

IF: Testing a Condition

The IF function tests a condition and returns a value based on whether the condition is true or false.

Syntax: **How to Test a Condition**

```
IF(test, val1, val2)
```

where:

test

Condition

Is an SQL search condition, which evaluates to true or false.

val1, val2

Are expressions of compatible types.

This function returns a value of the type of val1 and val2. If test is true, val1 is returned, otherwise val2 is returned.

Example: **Testing a Condition**

This example tests COUNTRY. If the value is ENGLAND, it returns LONDON. Otherwise, it returns PARIS.

```
IF(COUNTRY = 'ENGLAND', 'LONDON', 'PARIS') =
   'LONDON'   if COUNTRY is 'ENGLAND'
   'PARIS'    otherwise.
```

This example tests COUNTRY. If the value is ENGLAND, it returns LONDON. If the value is FRANCE, it returns PARIS. Otherwise, it returns ROME.

```
IF(COUNTRY = 'ENGLAND', 'LONDON',
   IF(COUNTRY = 'FRANCE', 'PARIS', 'ROME')) =
   'LONDON'   if COUNTRY is 'ENGLAND'
   'PARIS'    if COUNTRY = 'FRANCE'
   'ROME'     otherwise.
```

LEAST: Returning the Minimum Value From a List of Arguments

LEAST returns the minimum value from a list of arguments.

Syntax: **How to Return the Minimum Value From a List of Arguments**

```
LEAST(arg1, arg2, ...)
```

where:

arg1, arg2, ...

Numeric

Is a list of numeric arguments, which can be fields or literals.

Example: Returning the Minimum Value From a List of Arguments

LEAST returns either the value of the ED_HRS field, or the constant 30, whichever is lower:

```
GREATEST(ED_HRS, 30)
```

For ED_HRS = 45.00, the result is 30.00.

For ED_HRS = 25.00, the result is 25.00.

LENGTH: Obtaining the Physical Length of a Data Item

The LENGTH function returns the actual length in memory of a data item.

Syntax: How to Obtain the Physical Length of a Data Item

```
LENGTH(arg)
```

where:

arg

Any type

Is the length of the argument. It can be between 1 and 16 bytes.

This function returns an integer value.

Example: Obtaining the Physical Length of a Data Item

LENGTH returns the length in memory of a data item. This example,

```
LENGTH('abcdef')
```

returns 6.

This example,

```
LENGTH(3)
```

returns 4.

USER: Returning the User ID of the Connected User

USER returns the connected user ID.

Syntax: How to Return the User ID of the Connected User

```
USER()
```

Example: Returning the User ID of the Connected User

For the user with user ID USER01, USER() returns the following:

```
USER01
```

VALUE: Coalescing Data Values

Note: The SQL function VALUE is not supported. Instead, use the SQL operator COALESCE. For more information see *COALESCE: Coalescing Data Values* on page 469.

Chapter 24

SQL Operators

SQL operators are used to evaluate expressions.

In this chapter:

CASE: SQL Case Operator

The CASE operator allows a value to be computed depending on the values of expressions or the truth or falsity of conditions.

Syntax:

How to Use the SQL Case Operator

In the first format below the value of *test-expr* is compared to *value-expr-1, ..., value-expr-n* in turn:

- If any of these match, the value of the result is the corresponding *result-expr*.
- If there are no matches and the ELSE clause is present, the result is *else-expr*.
- If there are no matches and the ELSE clause is not present, the result is NULL.

In the second format below the values of *cond-1, ..., cond-n* are evaluated in turn.

- If any of these are true, the value of the result is the corresponding *result-expr*.
- If no conditions are true and the ELSE clause is present, the result is *else-expr*.

- ❏ If no conditions are true and the ELSE clause is not present, the result is NULL.

Format 1

```
CASE test-expr
   WHEN value-expr-1 THEN result-expr-1
   . . .
   WHEN value-expr-n THEN result-expr-n
   [ ELSE else-expr ]
END
```

Format 2

```
CASE
   WHEN cond-1 THEN result-expr-1
   . . .
   WHEN cond-n THEN result-expr-n
   [ ELSE else-expr ]
END
```

where:

test-expr

Any type

Is the value to be tested in Format 1.

value-expr1, ... , value-expr-n

Any type of compatible with *test-expr*.

Are the values *test-expr* is tested against in Format 1.

result-expr1, ... , result-expr-n

Any type

Are the values that become the result value if:

- ❏ The corresponding *value-expr* matches *test-expr* (Format 1).

or

- ❏ The corresponding *cond* is true (Format 2).

The result expressions must all have a compatible type.

cond-1, ..., cond-n

Condition

Are conditions that are tested in Format 2.

else-expr

Any type

Is the value of the result if no matches are found. Its type must be compatible with the result expressions.

This operator returns the compatible type of the result expressions.

Example: Using the SQL Case Operator

CASE returns values based on expressions. This example,

```
CASE COUNTRY
   WHEN 'ENGLAND' THEN 'LONDON'
   WHEN 'FRANCE' THEN 'PARIS'
   WHEN 'ITALY' THEN 'ROME'
   ELSE 'UNKNOWN'
END
```

returns LONDON when the value is ENGLAND, PARIS when the value is FRANCE, ROME when the value is ITALY, and UNKNOWN when there is no match.

COALESCE: Coalescing Data Values

The COALESCE operator can take 2 or more arguments. The first argument that is not NULL is returned. If all arguments are NULL, NULL is returned.

Syntax: How to Coalesce Data Values

```
COALESCE(arg1, arg2, [ ... argn ])
```

where:

arg1, arg2, ..., argn

Any type

Are data values. The types of the arguments must be compatible.

This operator returns the compatible type of the arguments.

Example: Coalescing Data Values

This example,

```
COALESCE('A', 'B')
```

return A.

This example,

```
COALESCE(NULL, 'B')
```

return B.

This example,

```
COALESCE(NULL, NULL)
```

return NULL.

EXISTS: Testing If a Subquery Returns One or More Rows

EXISTS can be used in a WHERE predicate to test whether a SUB-SELECT returns any data. If any rows are returned, EXISTS evaluates as true.

Syntax: How to Test If a Subquery Returns One or More Rows

```
SELECT ... WHERE EXISTS(SELECT * FROM lookup_mfd SQ [WHERE condition])
```

where:

lookup_mfd

Is the lookup Master File for the SUB-SELECT statement.

condition

Is the condition for the subquery.

Example: Testing If a Subquery Returns One or More Rows

The following SELECT statement counts the distinct customer IDs in WF_RETAIL_SALES if the first name supplied by the user exists in WF_RETAIL_CUSTOMERS.

```
SELECT  COUNT(DISTINCT(T1.ID_CUSTOMER)) FROM WF_RETAIL_SALES SQ
  WHERE EXISTS(SELECT * FROM WF_RETAIL_CUSTOMER SQ WHERE SQ.FIRSTNAME='&FN')
```

For Abbey, the result is 377923.

For Kathi, the result is 0.

IN: Determining Whether a Column Value Matches a Value in a List

The IN operator enables you to select a row from a data source by matching a column value against a list of acceptable values. If the test value is found on the list, that row is selected and returned for output.

Syntax: How to Determine Whether a Column Value Matches a Value in a List

```
WHERE test_exp IN (exp1, exp2, ...);
```

where:

test_exp

Is the column value to test against the list.

exp1, *exp2*, ...

Is the list of values for matching against the test value.

Example: Determining Whether a Value Matches a Value in a List

The following query displays start station names in Kings County and Queens County:

```
SQL
SELECT
     T1.START_STATION_NAME,
     T2.COUNTY
FROM
   (station_zip T2
     INNER JOIN
    citibike_tripdata T1
       ON
      T2.STATION_ID = T1.START_STATION_ID )
        WHERE T2.COUNTY IN('Kings County', 'Queens County')
;
END
```

Station *Clinton Ave & Flushing Ave* is selected because it is in Kings County.

Station *W 16 St & The High Line* is not selected because it is in New York County.

IN: Determining Whether Specified Column Values Match a Value Returned by a Subquery

The IN operator enables you to select a row from a data source by matching a column value against values returned by a subquery. If the test value is found in the values returned by the subquery, that row is selected and returned for output.

Syntax: How to Determine Whether a Column Value Matches a Value Returned by a Subquery

```
SELECT ...
WHERE (test_exp1[, test_exp2])
    IN (SELECT exp1, exp2
    FROM synonym [WHERE condition]);
```

where:

test_exp1[, *test_exp2*]

Are the column values to test against the output of the subquery.

exp1, *exp2*, ...

Are the list of values for matching against the test values.

synonym

Is the name of the synonym to be used in the subquery.

condition

Is a WHERE predicate for the subquery.

Example: Determining Whether a Value Matches a Value Returned by a Subquery

The following query displays start station names in Kings County:

```
SQL
SELECT
     T1.START_STATION_NAME
FROM
    citibike_tripdata T1
WHERE T1.START_STATION_ID IN (SELECT T2.STATION_ID FROM
citibike.station_zip T2
      WHERE T2.COUNTY = 'Kings County' )
;
END
```

Station *Clinton Ave & Flushing Ave* is selected because it is in Kings County.

Station *W 16 St & The High Line* is not selected because it is in New York County.

NULLIF: NULLIF Operator

The NULLIF operator returns NULL if its two arguments are equal. Otherwise, the first argument is returned.

Syntax: How to Use the NULLIF Operator

```
NULLIF(arg1, arg2)
```

where:

arg1, arg2

Any type

Are data values. The types of the two arguments must be compatible.

This operator returns the compatible type of the arguments.

Example: Using the NULLIF Operator

NULLIF operator returns NULL if two values are equal. This example,

```
NULLIF(IDNUM, -1)
```

returns NULL if the identification number is -1, otherwise it returns the number.

SELECT: Returning a Column Value Using a Subquery

SELECT returns a column value from a lookup file using an SQL subquery. The SUB-SELECT can only return a single value.

Syntax: How to Return a Column Value Using a Subquery

```
SELECT (SELECT [aggregation](SQ.lookup_result)
         FROM lookup_mfd SQ [WHERE condition])
         AS colname FROM original_mfd T1
```

where:

aggregation

Is an aggregation operator to apply to the lookup value. Can be one of the following operators:

- **AVG(***field***)** calculates the average.
- **AVG(DISTINCT** *field***)** calculates the average of distinct values.
- **MAX(***field***)** calculates the maximum.
- **MIN(***field***)** calculates the minimum.
- **COUNT(***field***)** calculates the count.
- **COUNT(DISTINCT** *field***)** calculates the count of distinct values.
- **SUM(***field***)** calculates the sum.
- **SUM(DISTINCT** *field***)** calculates the sum of distinct values.

field

Is the name of the column to be aggregated.

lookup_result

Is the lookup field.

lookup_mfd

Is the lookup Master File for the SUB-SELECT statement.

condition

Is the condition for the subquery.

colname

Is the name of the field for the returned value.

original_mfd

Is the Master File for the original SELECT statement.

Example: Returning a Column Value Using a Subquery

The following SUB-SELECT statement returns the sum of DAYSDELAYED from WF_RETAIL_SHIPMENTS and returns it to the SELECT statement against WF_RETAIL_SALES as the field named DELAY.

```
SELECT (SELECT SUM(SQ.DAYSDELAYED) FROM WF_RETAIL_SHIPMENTS SQ) AS DELAY
FROM WF_RETAIL_SALES T1
```

The result is 411338.

The following SUB-SELECT statement returns the first name of employee ID 409 from WF_RETAIL_EMPLOYEE and returns it to the SELECT statement against WF_RETAIL_LABOR as the field named FN.

```
SELECT (SELECT SQ.FIRSTNAME FROM WF_RETAIL_EMPLOYEE SQ WHERE SQ.ID_EMPLOYEE
= 409) AS FN FROM WF_RETAIL_LABOR T1
```

The result is Marina.

Chapter 25

SQL Aggregation Functions

Aggregate functions compute an aggregate value on a set of rows and return a single value.

In this chapter:

ASQ: Calculating the Average Sum of Squares

ASQ calculates the Average Sum of Squares, which is a measure of the deviation in a set of numbers. It is the average of the sum of squared differences of each data point from the mean.

Syntax: How to Calculate the Average Sum of Squares

```
ASQ(field)
```

where:

field

Is a numeric field on which to calculate the result.

Example: Calculating the Average Sum of Squares

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following Quantity in Stock (QTY_IN_STOCK) values:

```
43,068
13,527
   199
10,758
 2,972
   990
12,707
60,073
 5,961
 2,300
 4,000
12,444
11,499
22,000
 1,990
21,000
33,000
```

ASQ calculates the average sum of squares of Quantity in Stock.

```
ASQ(QTY_IN_STOCK)
```

For 258,488, the result is 487,971,549.

AVG: Calculating the Average of a Field

AVG calculates the average of a field.

Syntax: How to Calculate the Average of a Field

```
AVG(field)
```

where:

field

Is a numeric field on which to calculate the result.

Example: Calculating the Average of a Field

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following Quantity in Stock (QTY_IN_STOCK) values:

```
43,068
13,527
   199
10,758
 2,972
   990
12,707
60,073
 5,961
 2,300
 4,000
12,444
11,499
22,000
 1,990
21,000
33,000
```

AVG calculates the average of Quantity in Stock.

```
AVG(QTY_IN_STOCK)
```

For 258488, the result is 15205.

AVG(DISTINCT): Calculating the Average of Distinct Values in a Field

AVG(DISTINCT) calculates the average of the distinct values in a field.

Syntax: How to Calculate the Average of the Distinct Values in a Field

```
AVG(DISTINCT field)
```

where:

field

Is a numeric field on which to calculate the result.

Example: Calculating the Average of a Field

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following sale price (PRICE) values:

```
179.00
399.00
199.00
999.00
899.00
249.00
279.00
399.00
319.00
399.00
349.00
129.00
109.00
169.00
 89.00
499.00
299.00
```

AVG(DISTINCT) calculates the average of the distinct values of PRICE.

```
AVG(DISTINCT T1.PRICE)
```

The result is 344.33, while the average of all values is 350.76.

COUNT: Counting the Occurrences of a Field

COUNT counts the occurrences of a field.

Syntax: How to Count the Occurrences of a Field

```
COUNT(field)
```

where:

field

Is a numeric field on which to calculate the result.

Example: Counting the Occurrences of a Field

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following Quantity in Stock (QTY_IN_STOCK) values:

```
43,068
13,527
   199
10,758
 2,972
   990
12,707
60,073
 5,961
 2,300
 4,000
12,444
11,499
22,000
 1,990
21,000
33,000
```

COUNT counts the number of occurrences of Quantity in Stock.

```
COUNT(QTY_IN_STOCK)
```

The result is 17.

COUNT(DISTINCT): Calculating the Count of Distinct Values in a Field

COUNT(DISTINCT) calculates the count of the distinct values in a field.

Syntax: How to Calculate the Count of the Distinct Values in a Field

```
COUNT(DISTINCT field)
```

where:

field

Is a numeric field on which to calculate the result.

Example: Calculating the Average of a Field

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following sale price (PRICE) values:

```
179.00
399.00
199.00
999.00
899.00
249.00
279.00
399.00
319.00
399.00
349.00
129.00
109.00
169.00
 89.00
499.00
299.00
```

COUNT(DISTINCT) calculates the average of the distinct values of PRICE.

```
COUNT(DISTINCT T1.PRICE)
```

The result is 15, while the count of all values is 17.

MAX: Returning the Maximum Value in a Field

MAX returns the maximum value in a field.

Syntax: How to Return the Maximum Value in a Field

```
MAX(field)
```

where:

field

Is a numeric field on which to calculate the result.

Example: Returning the Maximum Value in a Field

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following Quantity in Stock (QTY_IN_STOCK) values:

```
43,068
13,527
   199
10,758
 2,972
   990
12,707
60,073
 5,961
 2,300
 4,000
12,444
11,499
22,000
 1,990
21,000
33,000
```

MAX returns the maximum value of Quantity in Stock.

```
MAX(QTY_IN_STOCK)
```

The result is 60073.

MEDIAN: Calculating the Median of a Field

MEDIAN calculates the median of a field. The median is the middle (50th percentile) value or, if there is an even number of occurrences, the average of the two middle values.

Syntax: How to Calculate the Median of a Field

```
MEDIAN(field)
```

where:

field

Is a numeric field on which to calculate the result.

Example: Calculating the Median of a Field

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following Quantity in Stock (QTY_IN_STOCK) values:

```
43,068
13,527
   199
10,758
 2,972
   990
12,707
60,073
 5,961
 2,300
 4,000
12,444
11,499
22,000
 1,990
21,000
33,000
```

MEDIAN returns the median value of Quantity in Stock.

```
MEDIAN(QTY_IN_STOCK)
```

The result is 11499.

MIN: Returning the Minimum Value in a Field

MIN returns the minimum value in a field.

Syntax: How to Return the Minimum Value in a Field

```
MIN(field)
```

where:

field

Is a numeric field on which to calculate the result.

Example: Returning the Minimum Value in a Field

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following Quantity in Stock (QTY_IN_STOCK) values:

```
43,068
13,527
   199
10,758
 2,972
   990
12,707
60,073
 5,961
 2,300
 4,000
12,444
11,499
22,000
 1,990
21,000
33,000
```

MIN returns the minimum value of Quantity in Stock.

```
MIN(QTY_IN_STOCK)
```

The result is 199.

MODE: Calculating the Mode of a Field

MODE calculates the mode of a field field. The mode is the most common value.

Syntax: How to Calculate the Mode of a Field

```
MODE(field)
```

where:

field
Is a numeric field on which to calculate the result.

Example: Calculating the Mode of a Field

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following sale price (PRICE) values:

```
179.00
399.00
199.00
999.00
899.00
249.00
279.00
399.00
319.00
399.00
349.00
129.00
109.00
169.00
 89.00
499.00
299.00
```

MODE returns the mode of sale price.

```
MODE(PRICE)
```

The result is 399.

SUM: Calculating the Sum of a Field

SUM calculates the sum of a field.

Syntax: How to Calculate the Sum of a Field

```
SUM(field)
```

where:

field

Is a numeric field on which to calculate the result.

Example: Calculating the Sum of a Field

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following Quantity in Stock (QTY_IN_STOCK) values:

```
43,068
13,527
   199
10,758
 2,972
   990
12,707
60,073
 5,961
 2,300
 4,000
12,444
11,499
22,000
 1,990
21,000
33,000
```

SUM calculates the average of Quantity in Stock.

```
SUM(QTY_IN_STOCK)
```

The result is 258488.

SUM(DISTINCT): Calculating the Sum of Distinct Values in a Field

SUM(DISTINCT) calculates the sum of the distinct values in a field.

Syntax: How to Calculate the Sum of the Distinct Values in a Field

```
SUM(DISTINCT field)
```

where:

field

Is a numeric field on which to calculate the result.

Example: Calculating the Average of a Field

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following sale price (PRICE) values:

```
179.00
399.00
199.00
999.00
899.00
249.00
279.00
399.00
319.00
399.00
349.00
129.00
109.00
169.00
 89.00
499.00
299.00
```

SUM(DISTINCT) calculates the average of the distinct values of PRICE.

```
SUM(DISTINCT T1.PRICE)
```

The result is 5,165,00, while the sum of all values is 5,963.00.

Chapter 26

SQL Analytic Functions

Analytic functions compute an aggregate value based on a group of rows, called a partition. They return multiple rows for each group. For some of the functions, a sliding window of rows can be defined. The window determines the range of rows used to perform the calculations for the current row. An optional ORDER BY clause can affect the result of the calculation. Analytic functions can appear only in the select list or ORDER BY clause.

In this chapter:

AVG: Averaging Values Over a Group of Rows

AVG averages column values within a partition.

Syntax: How to Average Values Over a Group of Rows

```
AVG(exp) OVER([PARTITION BY part1[, part2 ...]]
     [ORDER BY exp1[, exp2 ...]] [window_frame_clause])
```

where:

exp

Is the numeric expression used in the average.

part1, part2 ...

Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as it changes the rows that are included in the sliding window on which the calculation is performed.

window_frame_clause

Defines the sliding window within each partition (starting row and ending row for the window). The window frame clause defines a frame around the current row within a partition, over which the analytic function is evaluated. Both physical window frames (defined by ROWS) and logical window frames (defined by RANGE) are allowed. It is your responsibility to know the syntax for your environment.

Basic syntax for the window frame clause follows:

```
{ROWS|RANGE}
{
  {UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}   |
  {BETWEEN boundary_start AND boundary_end}
}
```

The basic syntax for the start of the boundary is:

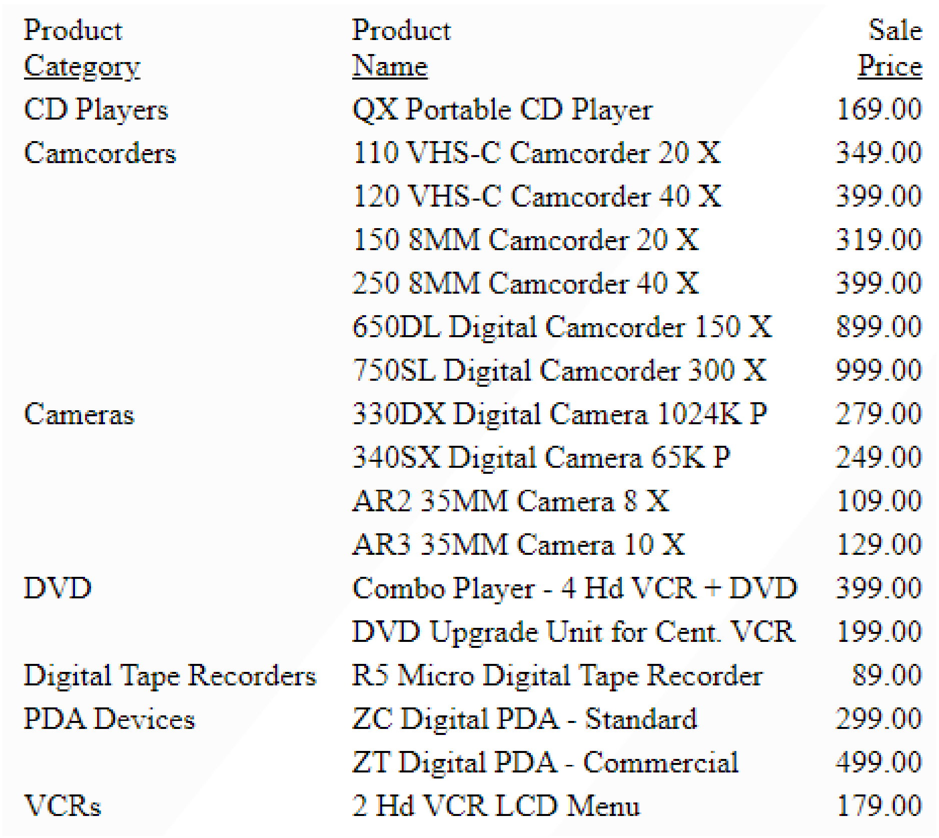
```
{UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}
```

The basic syntax for the end of the boundary is:

```
{UNBOUNDED FOLLOWING|numeric_expression {PRECEDING|FOLLOWING}|
      CURRENT ROW}
```

Example: Calculating an Average Within Product Category

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

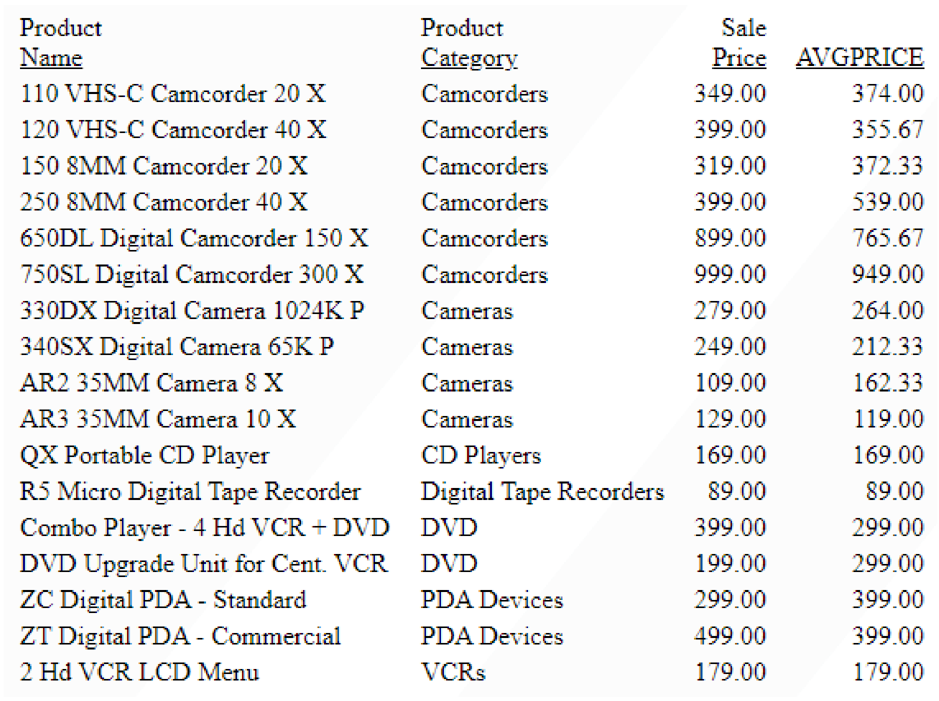


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request calculates the average price within product category, ordered by the product name, using a window that includes the current row and one row preceding and following.

```
SQL
SELECT
   PRODNAME,
   PRODCAT,
   PRICE,
   AVG(PRICE) OVER(PARTITION BY PRODCAT
      ORDER BY PRODNAME
      ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)  AS AVGPRICE
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Category	Sale Price	AVGPRICE
110 VHS-C Camcorder 20 X	Camcorders	349.00	374.00
120 VHS-C Camcorder 40 X	Camcorders	399.00	355.67
150 8MM Camcorder 20 X	Camcorders	319.00	372.33
250 8MM Camcorder 40 X	Camcorders	399.00	539.00
650DL Digital Camcorder 150 X	Camcorders	899.00	765.67
750SL Digital Camcorder 300 X	Camcorders	999.00	949.00
330DX Digital Camera 1024K P	Cameras	279.00	264.00
340SX Digital Camera 65K P	Cameras	249.00	212.33
AR2 35MM Camera 8 X	Cameras	109.00	162.33
AR3 35MM Camera 10 X	Cameras	129.00	119.00
QX Portable CD Player	CD Players	169.00	169.00
R5 Micro Digital Tape Recorder	Digital Tape Recorders	89.00	89.00
Combo Player - 4 Hd VCR + DVD	DVD	399.00	299.00
DVD Upgrade Unit for Cent. VCR	DVD	199.00	299.00
ZC Digital PDA - Standard	PDA Devices	299.00	399.00
ZT Digital PDA - Commercial	PDA Devices	499.00	399.00
2 Hd VCR LCD Menu	VCRs	179.00	179.00

The first value of AVGPRICE (374) is the average of the first two sale price values (current row, 349, and following row, 399), as there is no preceding row in the partition.

The second value of AVGPRICE (355.67) is the average of the sale prices in rows 1, 2, and 3 (349, 399, and 319).

This continues until the end of the partition (last Camcorders row), when the AVGPRICE value (949) is the average of the sale price in that row (999) and the preceding row (899).

For partitions that consist of one row, such as Digital Tape Recorders, the AVGPRICE value is the same as the sale price value, as there is no preceding or following row.

COUNT: Counting Values Over a Group of Rows

COUNT counts values over rows within a partition.

Syntax: How to Count Rows Within a Partition

```
COUNT(exp) OVER([PARTITION BY part1[, part2 ...]]
     [ORDER BY exp1[, exp2 ...]] [window_frame_clause])
```

where:

exp

Is the numeric expression used in the count.

part1, part2, ...

Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as it changes the rows that are included in the sliding window on which the calculation is performed.

window_frame_clause

Defines the sliding window within each partition (starting row and ending row for the window). The window frame clause defines a frame around the current row within a partition over which the analytic function is evaluated. Both physical window frames (defined by ROWS) and logical window frames (defined by RANGE) are allowed. It is your responsibility to know the syntax for your environment.

Basic syntax for the window frame clause follows:

```
{ROWS|RANGE}
{
  {UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}  |
  {BETWEEN boundary_start AND boundary_end}
}
```

The basic syntax for the start of the boundary is:

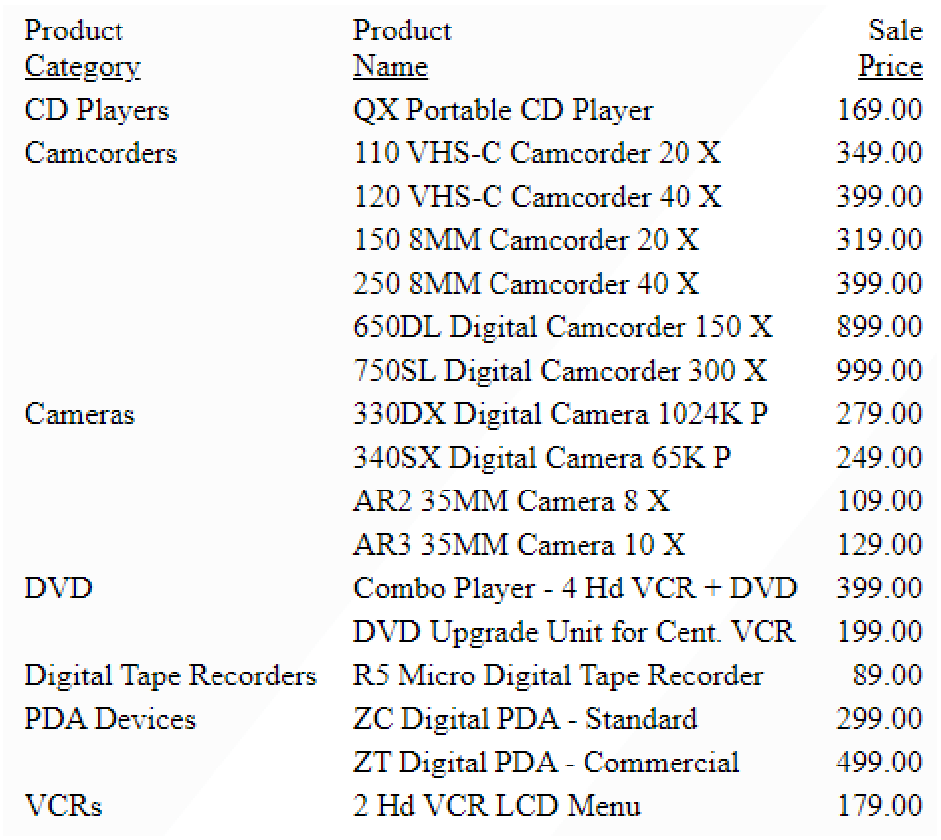
```
{UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}
```

The basic syntax for the end of the boundary is:

```
{UNBOUNDED FOLLOWING|numeric_expression {PRECEDING|FOLLOWING}|
      CURRENT ROW}
```

Example: **Counting Rows Within Product Category**

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

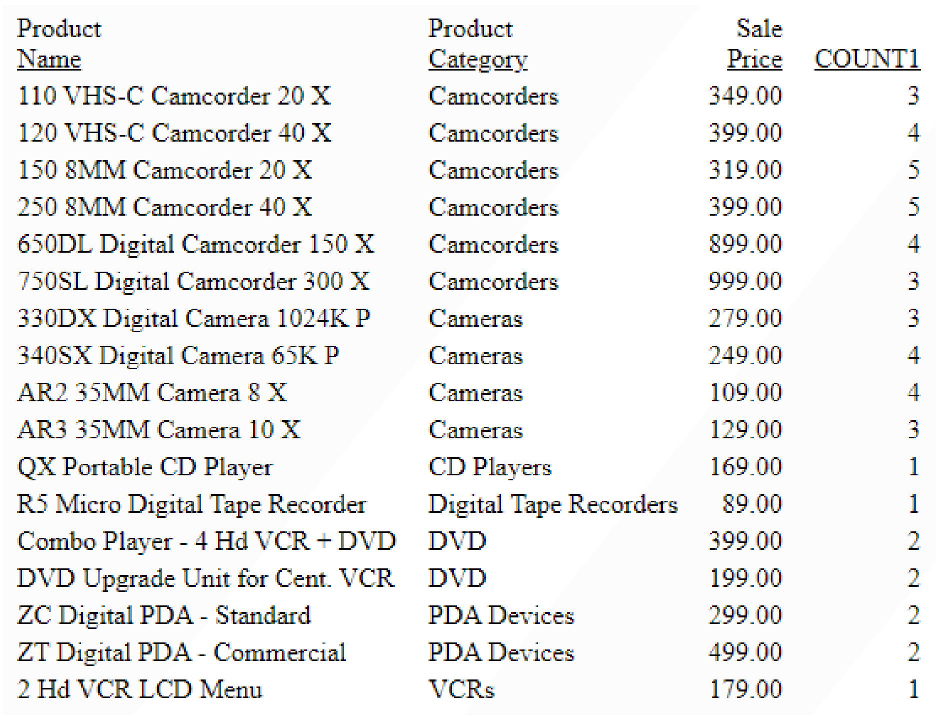


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request counts the number of price values within product category, ordered by the product name, using a window that includes the current row and two rows preceding and following.

```
SQL
SELECT
   PRODNAME,
   PRODCAT,
   PRICE,
   COUNT(PRICE) OVER(PARTITION BY PRODCAT
      ORDER BY PRODNAME
      ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING)  AS COUNT1
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Category	Sale Price	COUNT1
110 VHS-C Camcorder 20 X	Camcorders	349.00	3
120 VHS-C Camcorder 40 X	Camcorders	399.00	4
150 8MM Camcorder 20 X	Camcorders	319.00	5
250 8MM Camcorder 40 X	Camcorders	399.00	5
650DL Digital Camcorder 150 X	Camcorders	899.00	4
750SL Digital Camcorder 300 X	Camcorders	999.00	3
330DX Digital Camera 1024K P	Cameras	279.00	3
340SX Digital Camera 65K P	Cameras	249.00	4
AR2 35MM Camera 8 X	Cameras	109.00	4
AR3 35MM Camera 10 X	Cameras	129.00	3
QX Portable CD Player	CD Players	169.00	1
R5 Micro Digital Tape Recorder	Digital Tape Recorders	89.00	1
Combo Player - 4 Hd VCR + DVD	DVD	399.00	2
DVD Upgrade Unit for Cent. VCR	DVD	199.00	2
ZC Digital PDA - Standard	PDA Devices	299.00	2
ZT Digital PDA - Commercial	PDA Devices	499.00	2
2 Hd VCR LCD Menu	VCRs	179.00	1

The first value of COUNT1 (3) is the count of the first three sale price values (current row and two following rows), as there is no preceding row in the partition.

The second value of COUNT1 (4) is the count of the sale prices in rows 1, 2, 3, and 4.

This continues until the end of the partition (last Camcorders row), when the COUNT1 value (3) is the count of the sale price in that row and the two preceding rows.

For partitions that consist of one row, such as Digital Tape Recorders, the COUNT1 value is one (1), as there are no preceding or following rows.

DENSE_RANK: Assigning Rank Numbers With No Gaps

DENSE_RANK assigns rank numbers to rows, without any gaps. The PARTITION BY clause is optional, but the ORDER BY clause is required. If a partition is defined, the rank number restarts at 1 when the partition changes. The sliding window clause is not supported for DENSE_RANK.

Syntax: How to Assign Rank Numbers With No Gaps

```
DENSE_RANK() OVER([PARTITION BY part1[, part2 ...]]
     ORDER BY exp1[, exp2 ...] )
```

where:

part1, part2, ...

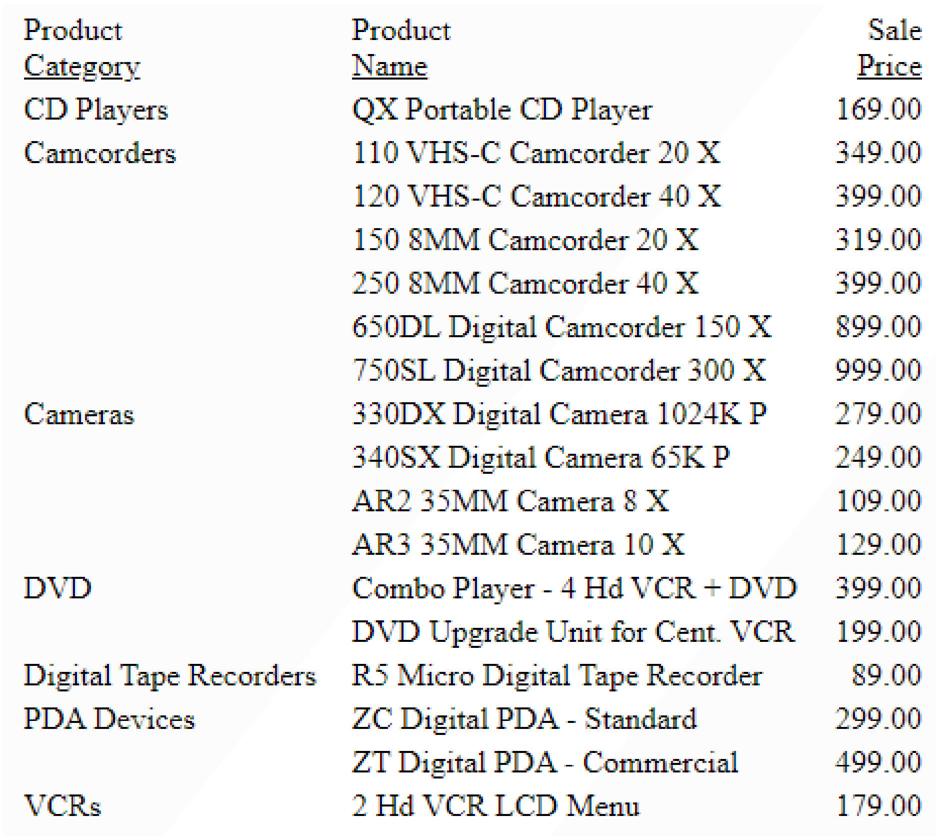
Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as ranks are assigned in row order.

Example: Assigning Rank Numbers With No Gaps

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

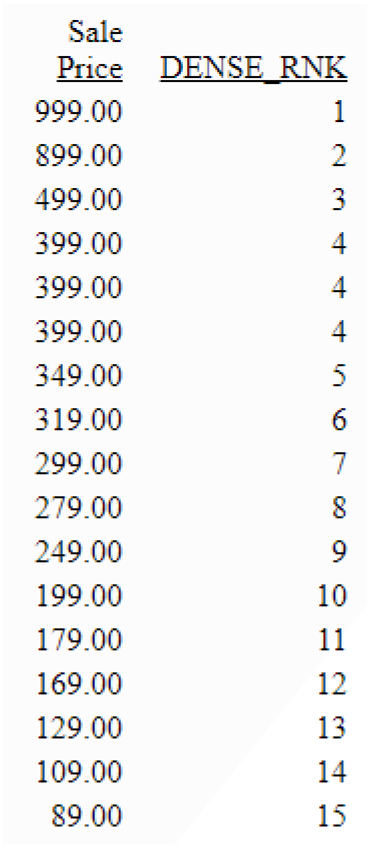


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request assigns dense rank numbers to all rows in decreasing order of price.

```
SQL
SELECT
  PRICE,
   DENSE_RANK() OVER( ORDER BY PRICE DESC )  AS DENSERNK ,
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Sale Price	DENSE_RNK
999.00	1
899.00	2
499.00	3
399.00	4
399.00	4
399.00	4
349.00	5
319.00	6
299.00	7
279.00	8
249.00	9
199.00	10
179.00	11
169.00	12
129.00	13
109.00	14
89.00	15

Three rows have the price 399.00. Each of those rows is assigned the rank 4, and the next row is assigned the rank 5 (with the RANK function, the ranks would go from 4 to 7, to account for the fact that there are three rows with rank 4).

FIRST_VALUE: Retrieving the First Result From an Ordered Set of Rows

FIRST_VALUE retrieves the first value in an ordered set of rows within a partition. An ORDER BY clause is required within the OVER clause, but the PARTITION BY clause is not required.

Syntax: How to Retrieve the First Value Within a Partition

```
FIRST_VALUE(exp) OVER([PARTITION BY part1[, part2 ...]]
     ORDER BY exp1[, exp2 ...] [window_frame_clause])
```

where:

exp

Is the expression used to calculate the result.

part1, part2, ...

Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as it changes the rows that are included in the sliding window on which the calculation is performed.

window_frame_clause

Defines the sliding window within each partition (starting row and ending row for the window). The window frame clause defines a frame around the current row within a partition over which the analytic function is evaluated. Both physical window frames (defined by ROWS) and logical window frames (defined by RANGE) are allowed. It is your responsibility to know the syntax for your environment.

Basic syntax for the window frame clause follows:

```
{ROWS|RANGE}
{
  {UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}  |
  {BETWEEN boundary_start AND boundary_end}
}
```

The basic syntax for the start of the boundary is:

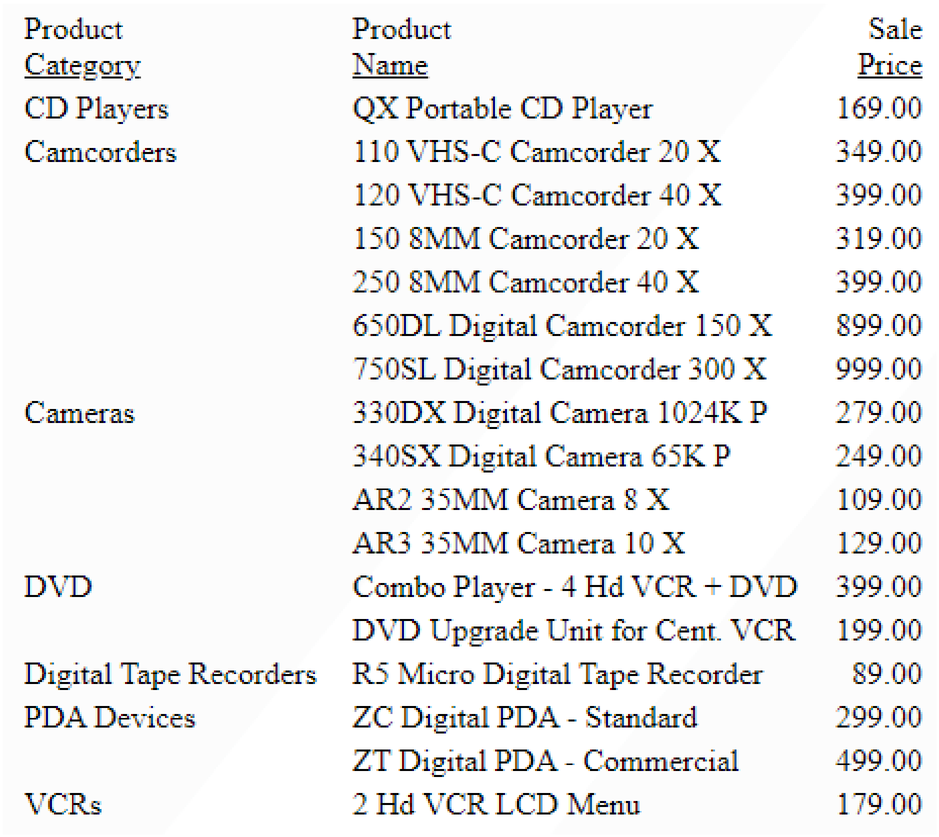
```
{UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}
```

The basic syntax for the end of the boundary is:

```
{UNBOUNDED FOLLOWING|numeric_expression {PRECEDING|FOLLOWING}|
     CURRENT ROW}
```

Example: Retrieving the First Value Within Product Category

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

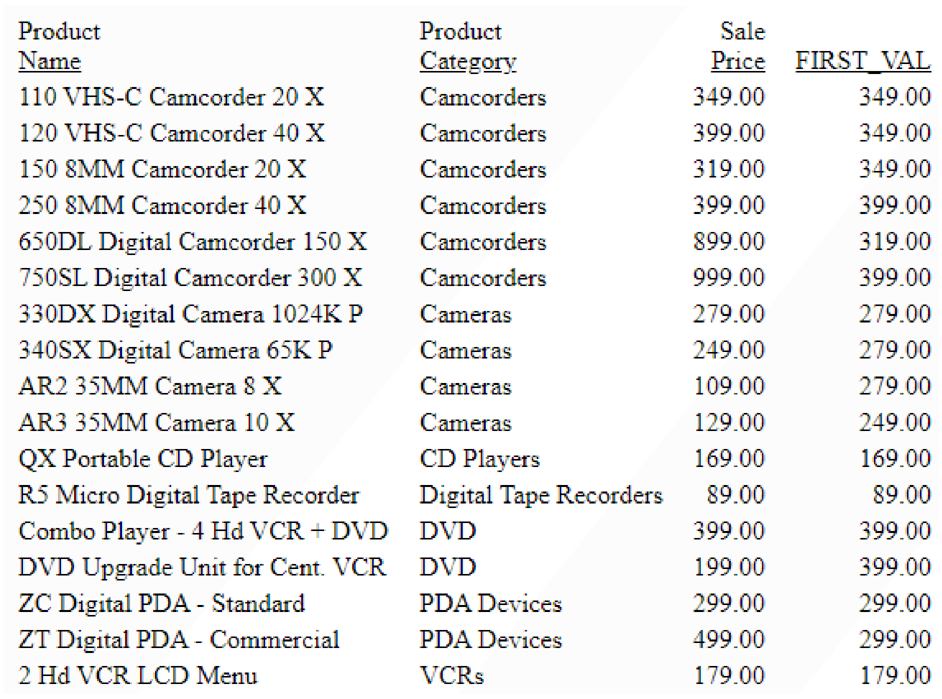


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request retrieves the first value of price within product category, ordered by the product name, using a window that includes the current row and two rows preceding and following.

```
SQL
SELECT
   PRODNAME,
   PRODCAT,
   PRICE,
   FIRST_VALUE(PRICE) OVER(PARTITION BY PRODCAT
      ORDER BY PRODNAME
      ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING)  AS COUNT1
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Category	Sale Price	FIRST_VAL
110 VHS-C Camcorder 20 X	Camcorders	349.00	349.00
120 VHS-C Camcorder 40 X	Camcorders	399.00	349.00
150 8MM Camcorder 20 X	Camcorders	319.00	349.00
250 8MM Camcorder 40 X	Camcorders	399.00	399.00
650DL Digital Camcorder 150 X	Camcorders	899.00	319.00
750SL Digital Camcorder 300 X	Camcorders	999.00	399.00
330DX Digital Camera 1024K P	Cameras	279.00	279.00
340SX Digital Camera 65K P	Cameras	249.00	279.00
AR2 35MM Camera 8 X	Cameras	109.00	279.00
AR3 35MM Camera 10 X	Cameras	129.00	249.00
QX Portable CD Player	CD Players	169.00	169.00
R5 Micro Digital Tape Recorder	Digital Tape Recorders	89.00	89.00
Combo Player - 4 Hd VCR + DVD	DVD	399.00	399.00
DVD Upgrade Unit for Cent. VCR	DVD	199.00	399.00
ZC Digital PDA - Standard	PDA Devices	299.00	299.00
ZT Digital PDA - Commercial	PDA Devices	499.00	299.00
2 Hd VCR LCD Menu	VCRs	179.00	179.00

The value of FIRST_VAL in row 1 (349.00) is the first of the sale price values in rows 1, 2, and 3 (current row and two following rows), as there is no preceding row in the partition.

The value of FIRST_VAL in row 2 (349.00) is the first of the sale prices in rows 1, 2, 3, and 4.

This continues until the beginning of the next partition (first Cameras row), when FIRST_VAL (279.00) is the first value of the sale price in that row and the two following rows.

For partitions that consist of one row, such as Digital Tape Recorders, the FIRST_VAL value is the same as the price in that row, as there are no preceding or following rows.

LAG: Retrieving Data From a Previous Row

LAG retrieves a value from a previous row given an offset from the current row. If a PARTITION BY clause is specified, the offset will only be used if it falls within the partition. If it does not, the default value specified in the function call will be returned. The sliding window clause is not supported for LAG.

Syntax: How to Retrieve Data From a Previous Row

```
LAG(exp ,[offset] [,default]) OVER([PARTITION BY part1[, part2 ...]]
     [ORDER BY exp1[, exp2 ...]] )
```

where:

exp

Is the value to be returned based on the specified offset.

offset

Is a positive integer value or expression that specifies the number of rows back from the current row from which to obtain a value. If not specified, the default is 1.

default

Is the value to return when offset is beyond the scope of the partition. If a default value is not specified, NULL is returned. The default value must be a data type that is compatible with *exp*.

part1, part2, ...

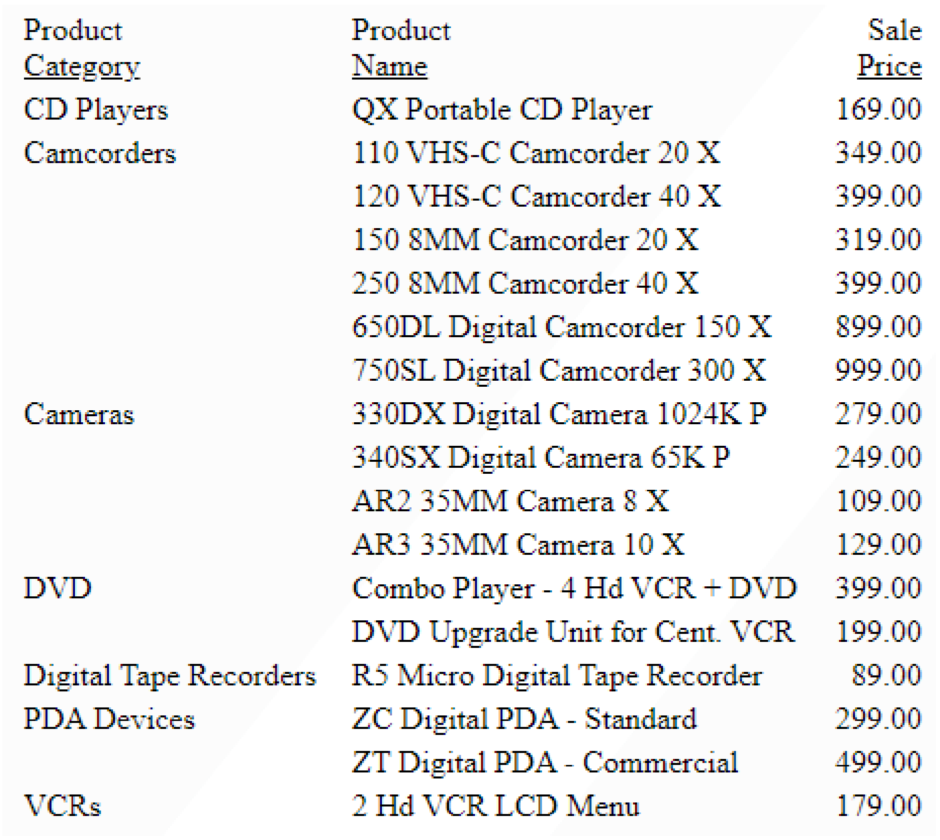
Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as ranks are assigned based on row order.

Example: Retrieving a Value From a Previous Row

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:



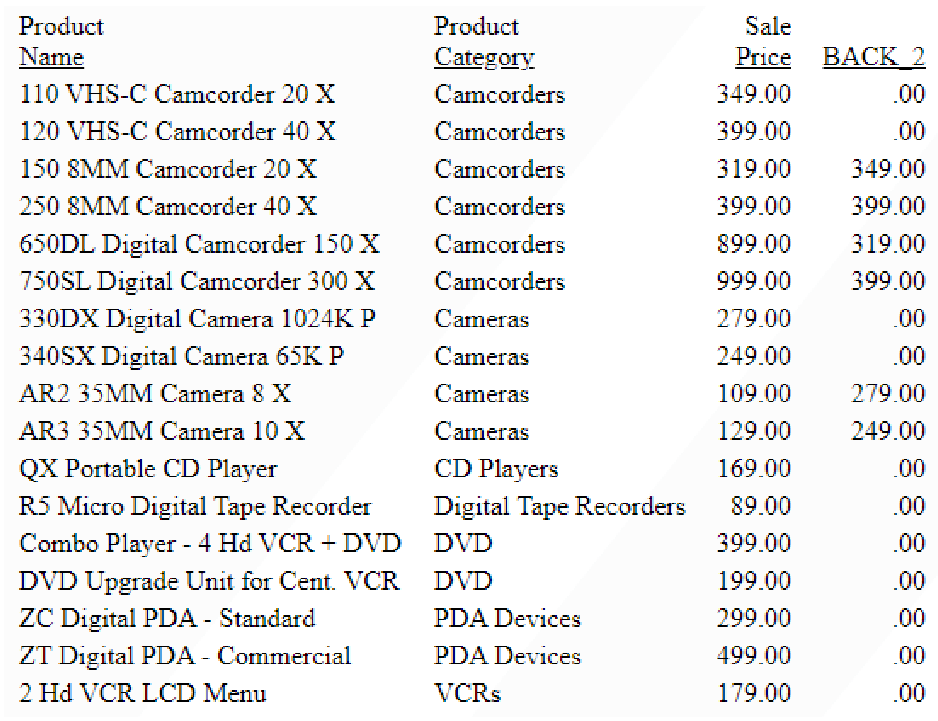
Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request retrieves the value of price two rows prior to the current row within the product category. If two rows back is not within the same partition, the default value (zero) is returned.

```
SSQL
SELECT
   PRODNAME,
   PRODCAT,
   PRICE,
   LAG(PRICE, 2, 0) OVER(
   PARTITION BY   PRODCAT
   ORDER BY  PRODNAME
   )  AS BACK_2

FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Category	Sale Price	BACK_2
110 VHS-C Camcorder 20 X	Camcorders	349.00	.00
120 VHS-C Camcorder 40 X	Camcorders	399.00	.00
150 8MM Camcorder 20 X	Camcorders	319.00	349.00
250 8MM Camcorder 40 X	Camcorders	399.00	399.00
650DL Digital Camcorder 150 X	Camcorders	899.00	319.00
750SL Digital Camcorder 300 X	Camcorders	999.00	399.00
330DX Digital Camera 1024K P	Cameras	279.00	.00
340SX Digital Camera 65K P	Cameras	249.00	.00
AR2 35MM Camera 8 X	Cameras	109.00	279.00
AR3 35MM Camera 10 X	Cameras	129.00	249.00
QX Portable CD Player	CD Players	169.00	.00
R5 Micro Digital Tape Recorder	Digital Tape Recorders	89.00	.00
Combo Player - 4 Hd VCR + DVD	DVD	399.00	.00
DVD Upgrade Unit for Cent. VCR	DVD	199.00	.00
ZC Digital PDA - Standard	PDA Devices	299.00	.00
ZT Digital PDA - Commercial	PDA Devices	499.00	.00
2 Hd VCR LCD Menu	VCRs	179.00	.00

In the first two rows, zero is returned because there is no value within the partition that is two rows back.

In the third row, the value is 349.00 because that is the Price value from row 1.

In the fourth row, the value is 399.00 because that is the Price value from row 2.

When a new partition starts (Cameras), the retrieval starts again, with zero returned for the first two rows.

LAST_VALUE: Retrieving the Last Result From an Ordered Set of Rows

LAST_VALUE retrieves the last value in an ordered set of rows within a partition. An ORDER BY clause is required within the OVER clause, but the PARTITION BY clause is not required.

Syntax: How to Retrieve the LAST Value Within a Partition

```
LAST_VALUE(exp) OVER([PARTITION BY part1[, part2 ...]]
     ORDER BY exp1[, exp2 ...] [window_frame_clause])
```

where:

exp

Is the expression used to calculate the result.

part1, part2, ...

Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as it changes the rows that are included in the sliding window on which the calculation is performed.

window_frame_clause

Defines the sliding window within each partition (starting row and ending row for the window). The window frame clause defines a frame around the current row within a partition over which the analytic function is evaluated. Both physical window frames (defined by ROWS) and logical window frames (defined by RANGE) are allowed. It is your responsibility to know the syntax for your environment.

Basic syntax for the window frame clause follows:

```
{ROWS|RANGE}
{
  {UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}  |
  {BETWEEN boundary_start AND boundary_end}
}
```

The basic syntax for the start of the boundary is:

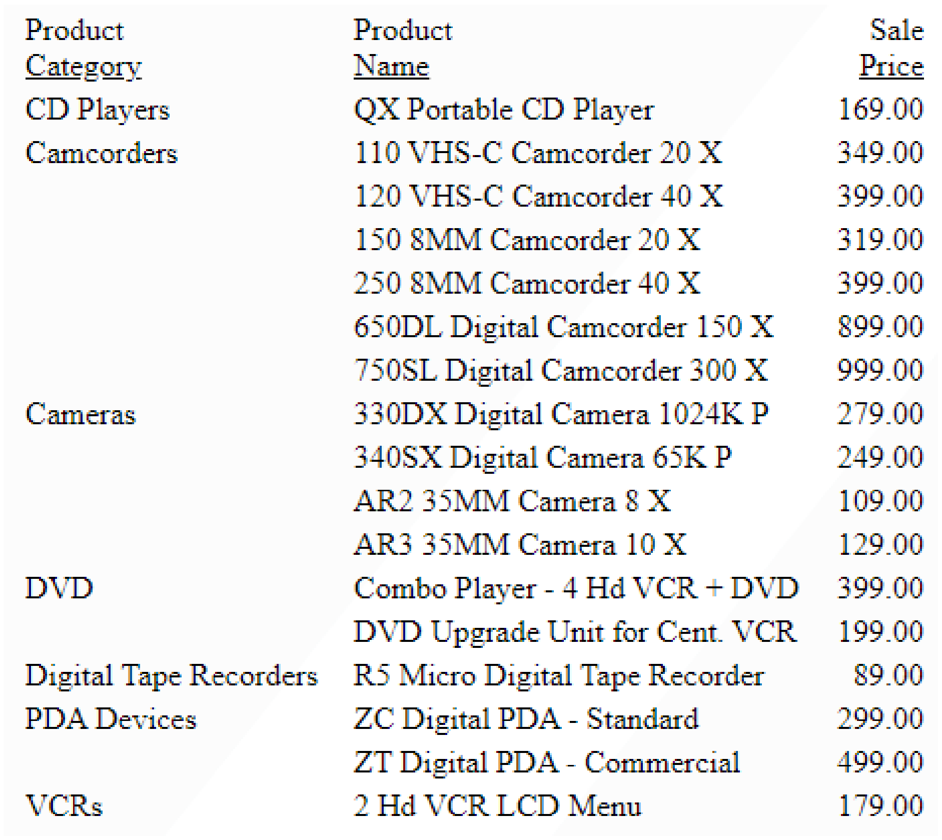
```
{UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}
```

The basic syntax for the end of the boundary is:

```
{UNBOUNDED FOLLOWING|numeric_expression {PRECEDING|FOLLOWING}|
      CURRENT ROW}
```

Example: Retrieving the Last Value Within Product Category

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

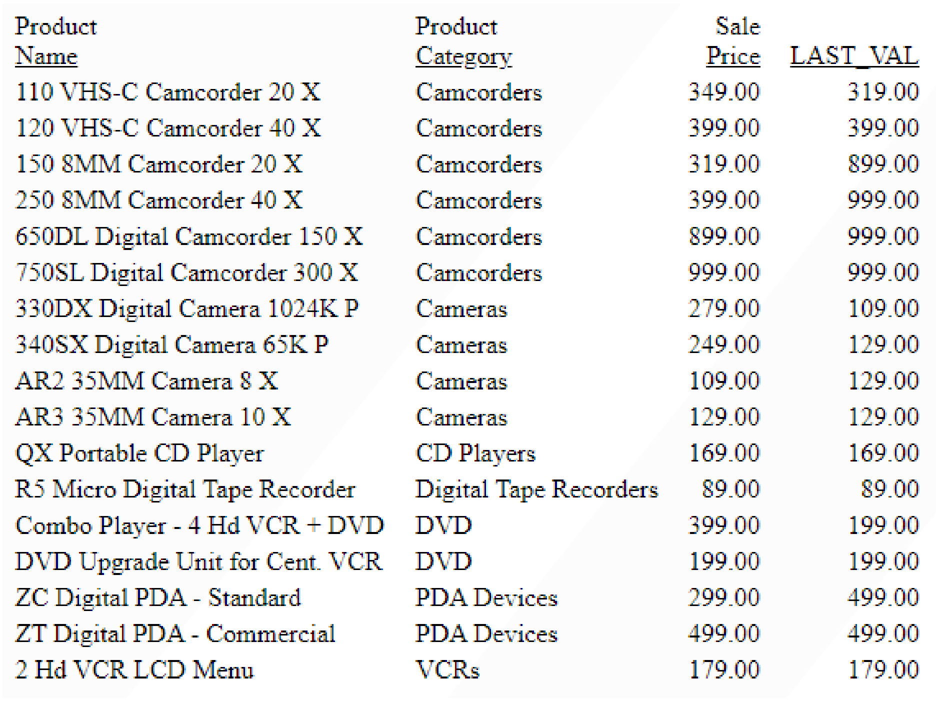


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request retrieves the last value of price within product category, ordered by the product name, using a window that includes the current row and two rows preceding and following.

```
SQL
SELECT
   PRODNAME,
   PRODCAT,
   PRICE,
   LAST_VALUE(PRICE) OVER(PARTITION BY PRODCAT
      ORDER BY PRODNAME
      ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING)  AS COUNT1
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Category	Sale Price	LAST_VAL
110 VHS-C Camcorder 20 X	Camcorders	349.00	319.00
120 VHS-C Camcorder 40 X	Camcorders	399.00	399.00
150 8MM Camcorder 20 X	Camcorders	319.00	899.00
250 8MM Camcorder 40 X	Camcorders	399.00	999.00
650DL Digital Camcorder 150 X	Camcorders	899.00	999.00
750SL Digital Camcorder 300 X	Camcorders	999.00	999.00
330DX Digital Camera 1024K P	Cameras	279.00	109.00
340SX Digital Camera 65K P	Cameras	249.00	129.00
AR2 35MM Camera 8 X	Cameras	109.00	129.00
AR3 35MM Camera 10 X	Cameras	129.00	129.00
QX Portable CD Player	CD Players	169.00	169.00
R5 Micro Digital Tape Recorder	Digital Tape Recorders	89.00	89.00
Combo Player - 4 Hd VCR + DVD	DVD	399.00	199.00
DVD Upgrade Unit for Cent. VCR	DVD	199.00	199.00
ZC Digital PDA - Standard	PDA Devices	299.00	499.00
ZT Digital PDA - Commercial	PDA Devices	499.00	499.00
2 Hd VCR LCD Menu	VCRs	179.00	179.00

The value of LAST_VAL in row 1 (319.00) is the last of the sale price values in rows 1, 2, and 3 (current row and two following rows), as there is no preceding row in the partition.

The value of LAST_VAL in row 2 (399.00) is the last of the sale prices in rows 1, 2, 3, and 4.

This continues until the beginning of the next partition (first Cameras row), when LAST_VAL (109.00) is the last value of the sale price in that row and the two following rows.

For partitions that consist of one row, such as Digital Tape Recorders, the LAST_VAL value is the same as the price in that row, as there are no preceding or following rows.

LEAD: Retrieving Data From a Subsequent Row

LEAD retrieves a value from a subsequent row given an offset from the current row. If a PARTITION BY clause is specified, the offset will only be used if it falls within the partition. If it does not, the default value specified in the function call will be returned. The sliding window clause is not supported for LEAD.

Syntax: How to Retrieve Data From a Subsequent Row

```
LEAD(exp ,[offset] [,default]) OVER([PARTITION BY part1[, part2 ...]]
     [ORDER BY exp1[, exp2 ...]] )
```

where:

exp

Is the value to be returned based on the specified offset.

offset

Is a positive integer value or expression that specifies the number of rows forward from the current row from which to obtain a value. If not specified, the default is 1.

default

Is the value to return when offset is beyond the scope of the partition. If a default value is not specified, NULL is returned. The default value must be a data type that is compatible with *exp*.

part1, part2, ...

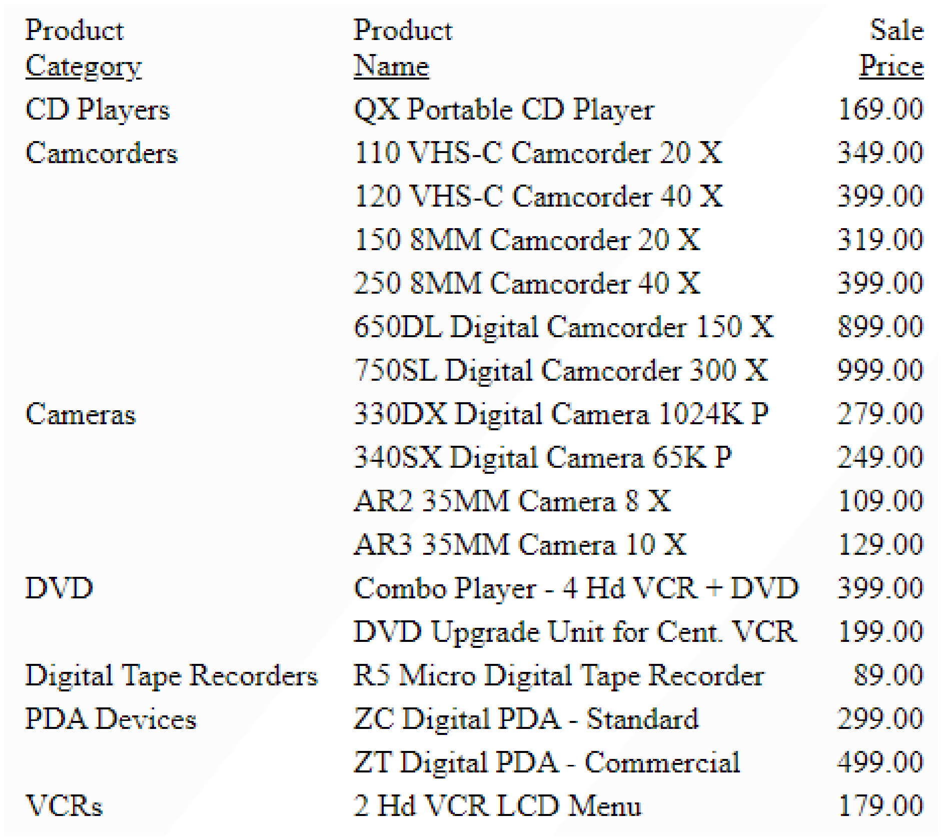
Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as ranks are assigned based on row order.

Example: **Retrieving a Value From a Previous Row**

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:



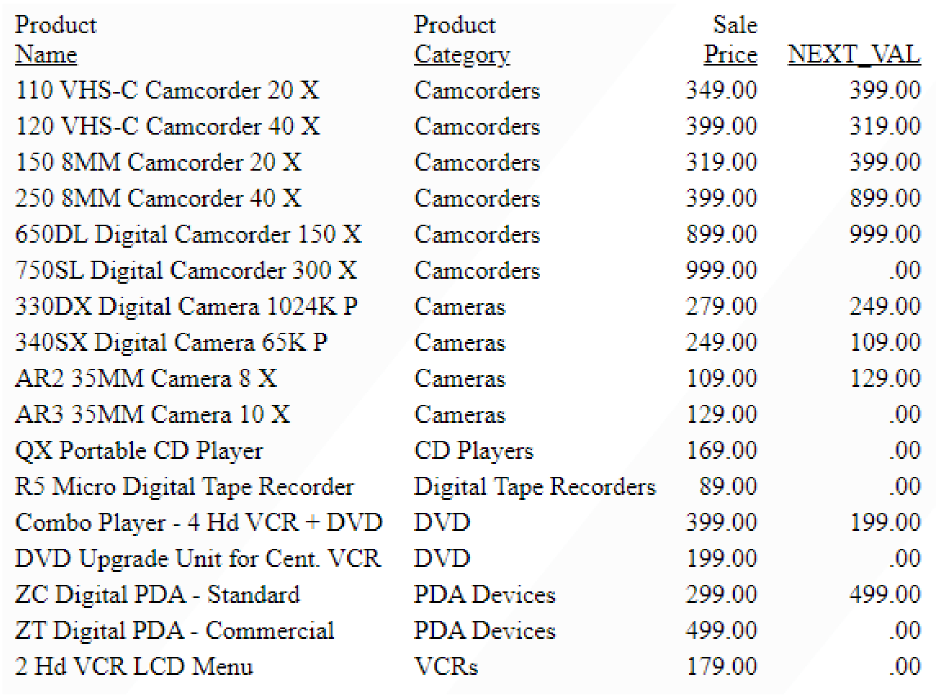
Product	Product	Sale
Category	Name	Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request retrieves the value of price one row following the current row within the product category. If the next row forward is not within the same partition, the default value (zero) is returned.

```
SSQL
SELECT
   PRODNAME,
   PRODCAT,
   PRICE,
   LEAD(PRICE, 1, 0) OVER(
   PARTITION BY   PRODCAT
   ORDER BY  PRODNAME
   )  AS NEXT_VAL

FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Category	Sale Price	NEXT_VAL
110 VHS-C Camcorder 20 X	Camcorders	349.00	399.00
120 VHS-C Camcorder 40 X	Camcorders	399.00	319.00
150 8MM Camcorder 20 X	Camcorders	319.00	399.00
250 8MM Camcorder 40 X	Camcorders	399.00	899.00
650DL Digital Camcorder 150 X	Camcorders	899.00	999.00
750SL Digital Camcorder 300 X	Camcorders	999.00	.00
330DX Digital Camera 1024K P	Cameras	279.00	249.00
340SX Digital Camera 65K P	Cameras	249.00	109.00
AR2 35MM Camera 8 X	Cameras	109.00	129.00
AR3 35MM Camera 10 X	Cameras	129.00	.00
QX Portable CD Player	CD Players	169.00	.00
R5 Micro Digital Tape Recorder	Digital Tape Recorders	89.00	.00
Combo Player - 4 Hd VCR + DVD	DVD	399.00	199.00
DVD Upgrade Unit for Cent. VCR	DVD	199.00	.00
ZC Digital PDA - Standard	PDA Devices	299.00	499.00
ZT Digital PDA - Commercial	PDA Devices	499.00	.00
2 Hd VCR LCD Menu	VCRs	179.00	.00

In the first row, the value from row 2 (399.00) is returned.

In the second row, the value from row 3 (319.00) is returned.

On the last row of the partition, zero is returned because the next row is not within the partition.

MAX: Calculating the Maximum Over a Group of Rows

MAX calculates the maximum column value within a partition.

Syntax: **How to Calculate the Maximum Over a Group of Rows**

```
MAX(exp) OVER([PARTITION BY part1[, part2 ...]]
     [ORDER BY exp1[, exp2 ...]] [window_frame_clause])
```

where:

exp

Is the numeric expression used in the calculation.

part1, part2 ...

Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as it changes the rows that are included in the sliding window on which the calculation is performed.

window_frame_clause

Defines the sliding window within each partition (starting row and ending row for the window). The window frame clause defines a frame around the current row within a partition over which the analytic function is evaluated. Both physical window frames (defined by ROWS) and logical window frames (defined by RANGE) are allowed. It is your responsibility to know the syntax for your environment.

Basic syntax for the window frame clause follows:

```
{ROWS|RANGE}
{
  {UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}   |
  {BETWEEN boundary_start AND boundary_end}
}
```

The basic syntax for the start of the boundary is:

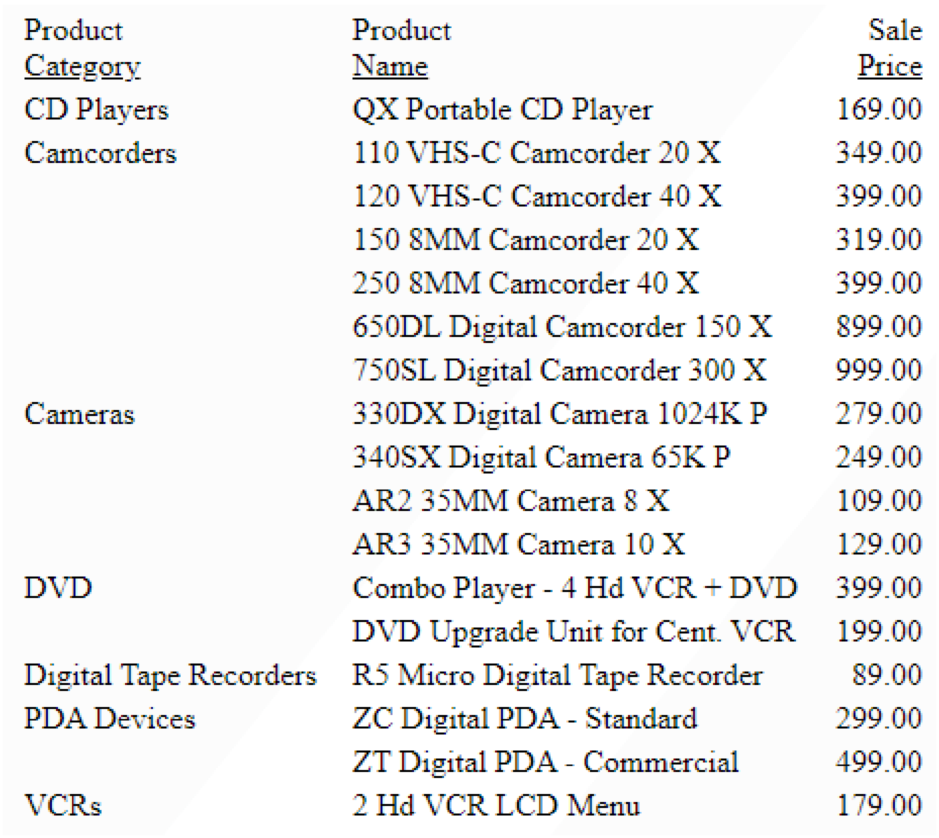
```
{UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}
```

The basic syntax for the end of the boundary is:

```
{UNBOUNDED FOLLOWING|numeric_expression {PRECEDING|FOLLOWING}|
     CURRENT ROW}
```

Example: Calculating the Maximum Within Product Category

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

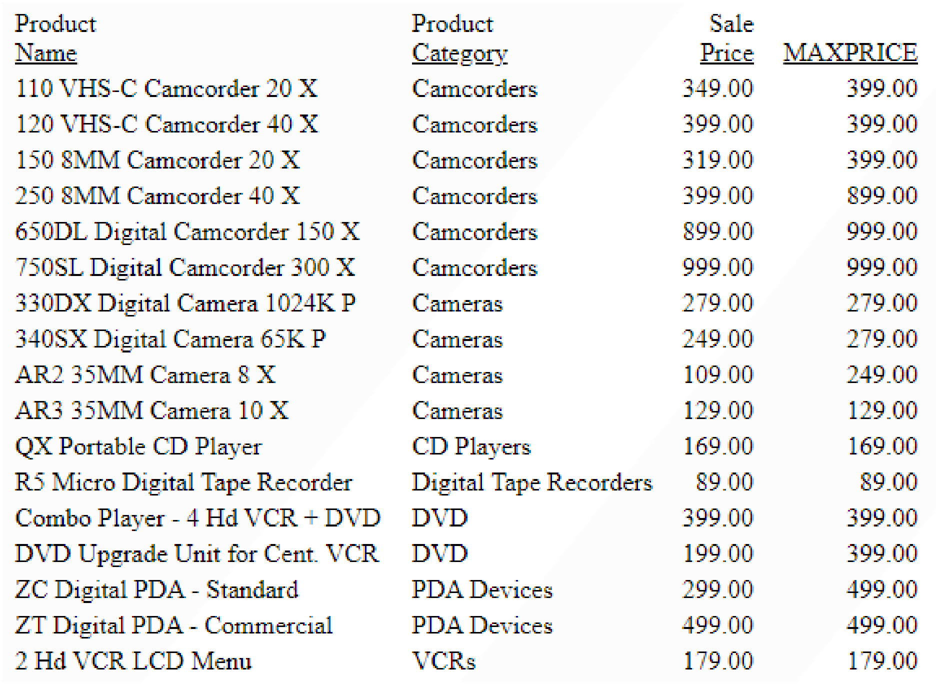


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request calculates the maximum price within product category, ordered by the product name, using a window that includes the current row and one row preceding and following.

```
SQL
SELECT
   PRODNAME,
   PRODCAT,
   PRICE,
   MAX(PRICE) OVER(PARTITION BY PRODCAT
      ORDER BY PRODNAME
      ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)  AS MAXPRICE
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Category	Sale Price	MAXPRICE
110 VHS-C Camcorder 20 X	Camcorders	349.00	399.00
120 VHS-C Camcorder 40 X	Camcorders	399.00	399.00
150 8MM Camcorder 20 X	Camcorders	319.00	399.00
250 8MM Camcorder 40 X	Camcorders	399.00	899.00
650DL Digital Camcorder 150 X	Camcorders	899.00	999.00
750SL Digital Camcorder 300 X	Camcorders	999.00	999.00
330DX Digital Camera 1024K P	Cameras	279.00	279.00
340SX Digital Camera 65K P	Cameras	249.00	279.00
AR2 35MM Camera 8 X	Cameras	109.00	249.00
AR3 35MM Camera 10 X	Cameras	129.00	129.00
QX Portable CD Player	CD Players	169.00	169.00
R5 Micro Digital Tape Recorder	Digital Tape Recorders	89.00	89.00
Combo Player - 4 Hd VCR + DVD	DVD	399.00	399.00
DVD Upgrade Unit for Cent. VCR	DVD	199.00	399.00
ZC Digital PDA - Standard	PDA Devices	299.00	499.00
ZT Digital PDA - Commercial	PDA Devices	499.00	499.00
2 Hd VCR LCD Menu	VCRs	179.00	179.00

The first value of MAXPRICE (399) is the maximum of the first two sale price values (current row, 349, and following row, 399), as there is no preceding row in the partition.

The second value of MAXPRICE (399) is the maximum of the sale prices in rows 1, 2, and 3 (349, 399, and 319).

This continues until the end of the partition (last Camcorders row), when the MAXPRICE value (999) is the maximum of the sale price in that row (999) and the preceding row (899).

For partitions that consist of one row, such as Digital Tape Recorders, the MAXPRICE value is the same as the sale price value, as there is no preceding or following row.

MEDIAN: Calculating the Median Over a Group of Rows

MEDIAN calculates the median column value within a partition.

Syntax: How to Calculate the Median Over a Group of Rows

```
MEDIAN(exp) OVER([PARTITION BY part1[, part2 ...]])
```

where:

exp

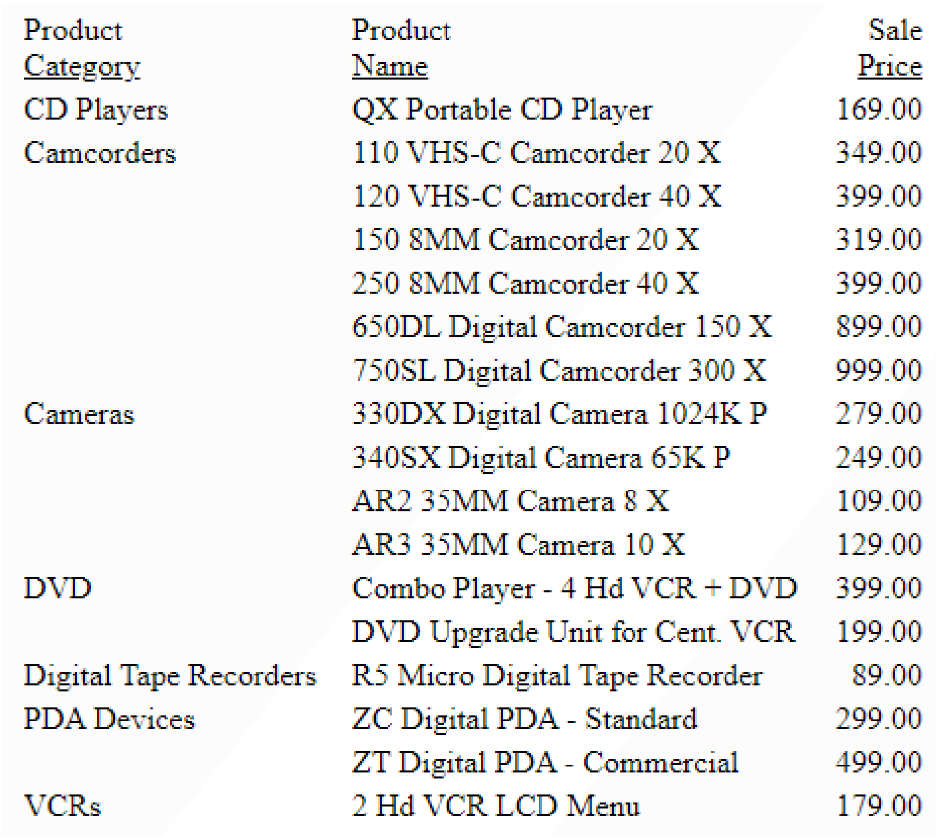
Is the numeric expression used in the calculation.

part1, part2 ...

Are partitioning columns or expressions.

Example: **Calculating the Median Within Product Category**

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

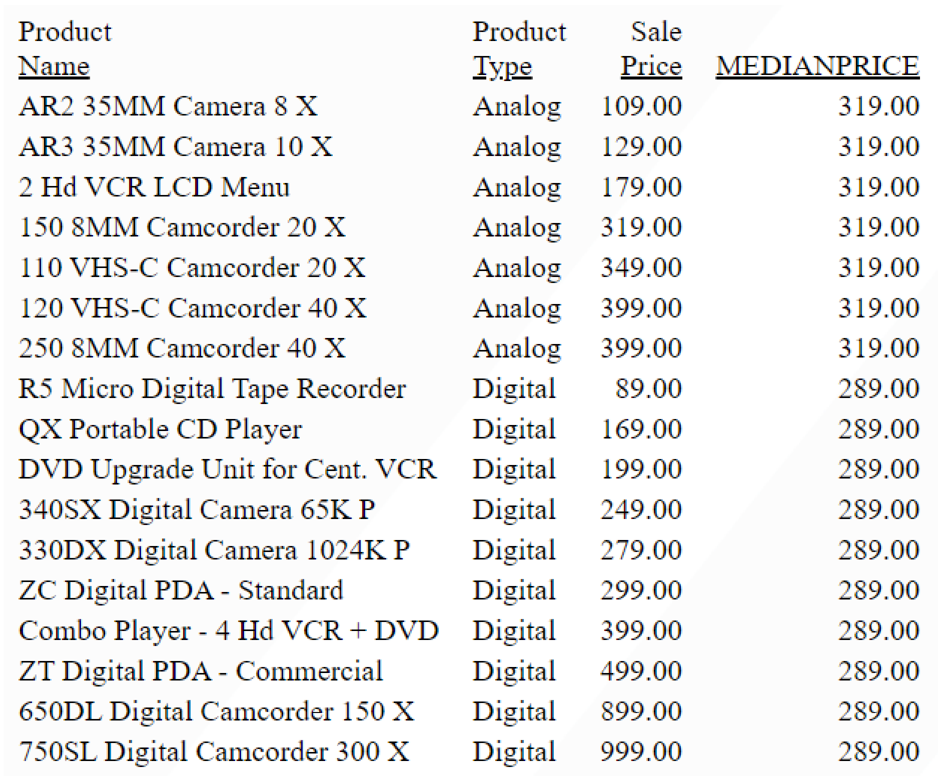


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request calculates the median price within product category, ordered by the product name, using a window that includes the current row and one row preceding and following.

```
SQL
SELECT
   PRODNAME,
   PRODTYPE,
   PRICE,
   MEDIAN(PRICE) OVER(PARTITION BY PRODTYPE) AS MEDIANPRICE
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Type	Sale Price	MEDIANPRICE
AR2 35MM Camera 8 X	Analog	109.00	319.00
AR3 35MM Camera 10 X	Analog	129.00	319.00
2 Hd VCR LCD Menu	Analog	179.00	319.00
150 8MM Camcorder 20 X	Analog	319.00	319.00
110 VHS-C Camcorder 20 X	Analog	349.00	319.00
120 VHS-C Camcorder 40 X	Analog	399.00	319.00
250 8MM Camcorder 40 X	Analog	399.00	319.00
R5 Micro Digital Tape Recorder	Digital	89.00	289.00
QX Portable CD Player	Digital	169.00	289.00
DVD Upgrade Unit for Cent. VCR	Digital	199.00	289.00
340SX Digital Camera 65K P	Digital	249.00	289.00
330DX Digital Camera 1024K P	Digital	279.00	289.00
ZC Digital PDA - Standard	Digital	299.00	289.00
Combo Player - 4 Hd VCR + DVD	Digital	399.00	289.00
ZT Digital PDA - Commercial	Digital	499.00	289.00
650DL Digital Camcorder 150 X	Digital	899.00	289.00
750SL Digital Camcorder 300 X	Digital	999.00	289.00

The first value of MEDIANPRICE (319) is the median of the sale prices for Analog. Since there are seven rows, the median is the middle one for all seven rows.

The next MEDIANPRICE (289) is for Digital. There are 10 rows, so the median is the average of the two middle values (279 and 299).

MIN: Calculating the Minimum Over a Group of Rows

MIN calculates the minimum column value within a partition.

Syntax: How to Calculate the Minimum Over a Group of Rows

```
MIN(exp) OVER([PARTITION BY part1[, part2 ...]]
     [ORDER BY exp1[, exp2 ...]] [window_frame_clause])
```

where:

exp

Is the numeric expression used in the calculation.

part1, part2 ...

Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as it changes the rows that are included in the sliding window on which the calculation is performed.

window_frame_clause

Defines the sliding window within each partition (starting row and ending row for the window). The window frame clause defines a frame around the current row within a partition over which the analytic function is evaluated. Both physical window frames (defined by ROWS) and logical window frames (defined by RANGE) are allowed. It is your responsibility to know the syntax for your environment.

Basic syntax for the window frame clause follows:

```
{ROWS|RANGE}
{
  {UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}  |
  {BETWEEN boundary_start AND boundary_end}
}
```

The basic syntax for the start of the boundary is:

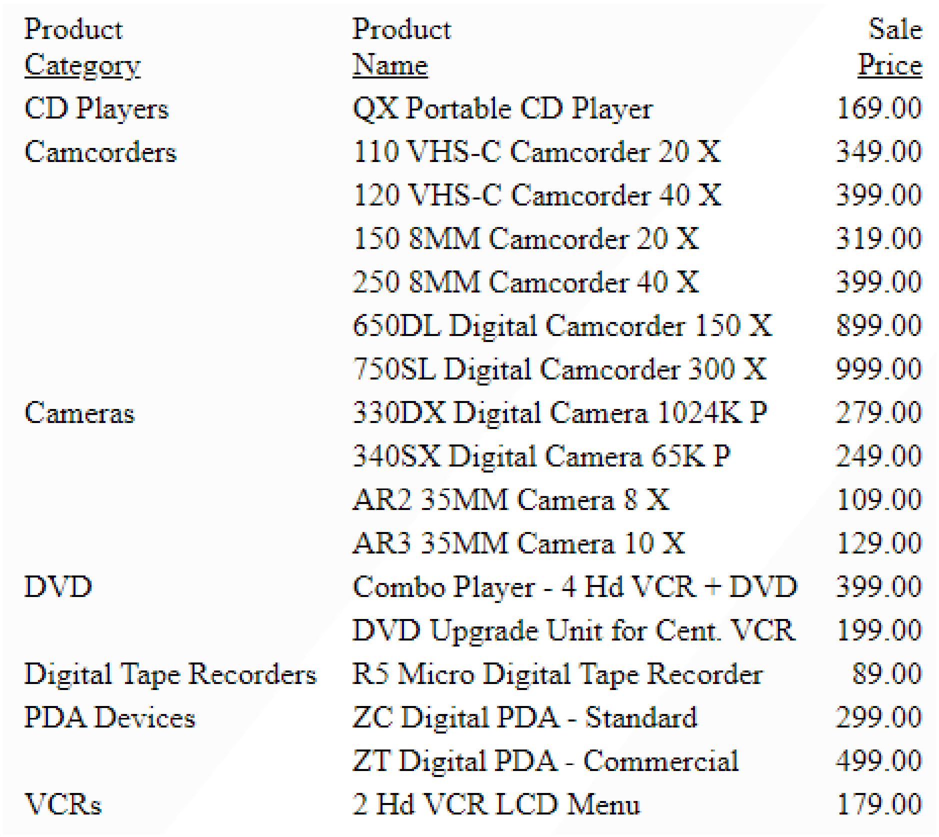
```
{UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}
```

The basic syntax for the end of the boundary is:

```
{UNBOUNDED FOLLOWING|numeric_expression {PRECEDING|FOLLOWING}|
      CURRENT ROW}
```

Example: Calculating the Minimum Within Product Category

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

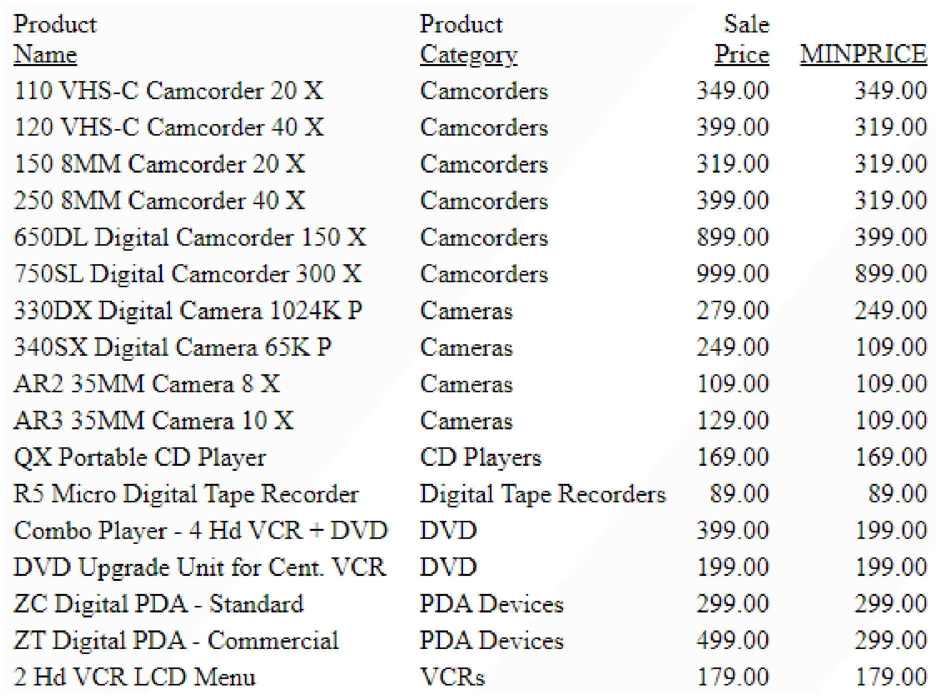


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request calculates the minimum price within product category, ordered by the product name, using a window that includes the current row and one row preceding and following.

```
SQL
SELECT
   PRODNAME,
   PRODCAT,
   PRICE,
   MIN(PRICE) OVER(PARTITION BY PRODCAT
      ORDER BY PRODNAME
      ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)  AS MINPRICE
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Category	Sale Price	MINPRICE
110 VHS-C Camcorder 20 X	Camcorders	349.00	349.00
120 VHS-C Camcorder 40 X	Camcorders	399.00	319.00
150 8MM Camcorder 20 X	Camcorders	319.00	319.00
250 8MM Camcorder 40 X	Camcorders	399.00	319.00
650DL Digital Camcorder 150 X	Camcorders	899.00	399.00
750SL Digital Camcorder 300 X	Camcorders	999.00	899.00
330DX Digital Camera 1024K P	Cameras	279.00	249.00
340SX Digital Camera 65K P	Cameras	249.00	109.00
AR2 35MM Camera 8 X	Cameras	109.00	109.00
AR3 35MM Camera 10 X	Cameras	129.00	109.00
QX Portable CD Player	CD Players	169.00	169.00
R5 Micro Digital Tape Recorder	Digital Tape Recorders	89.00	89.00
Combo Player - 4 Hd VCR + DVD	DVD	399.00	199.00
DVD Upgrade Unit for Cent. VCR	DVD	199.00	199.00
ZC Digital PDA - Standard	PDA Devices	299.00	299.00
ZT Digital PDA - Commercial	PDA Devices	499.00	299.00
2 Hd VCR LCD Menu	VCRs	179.00	179.00

The first value of MINPRICE (349) is the minimum of the first two sale price values (current row, 349, and following row, 399), as there is no preceding row in the partition.

The second value of MINPRICE (319) is the minimum of the sale prices in rows 1, 2, and 3 (349, 399, and 319).

This continues until the end of the partition (last Camcorders row), when the MINPRICE value (899) is the minimum of the sale price in that row (999) and the preceding row (899).

For partitions that consist of one row, such as Digital Tape Recorders, the MINPRICE value is the same as the sale price value, as there is no preceding or following row.

MODE: Calculating the Mode Over a Group of Rows

MODE calculates the mode column value within a partition.

Syntax: How to Calculate the Mode Over a Group of Rows

```
MODE(exp) OVER([PARTITION BY part1[, part2 ...]])
```

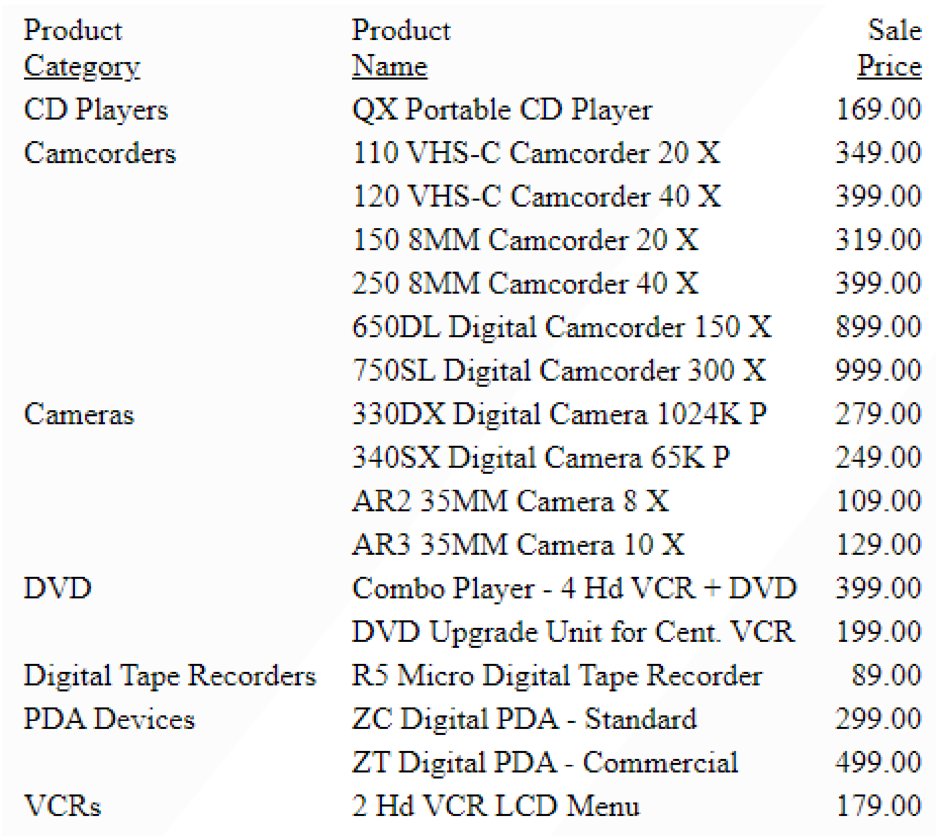
where:

exp
: Is the numeric expression used in the calculation.

part1, part2 ...
: Are partitioning columns or expressions.

Example: Calculating the Mode Within Product Category

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

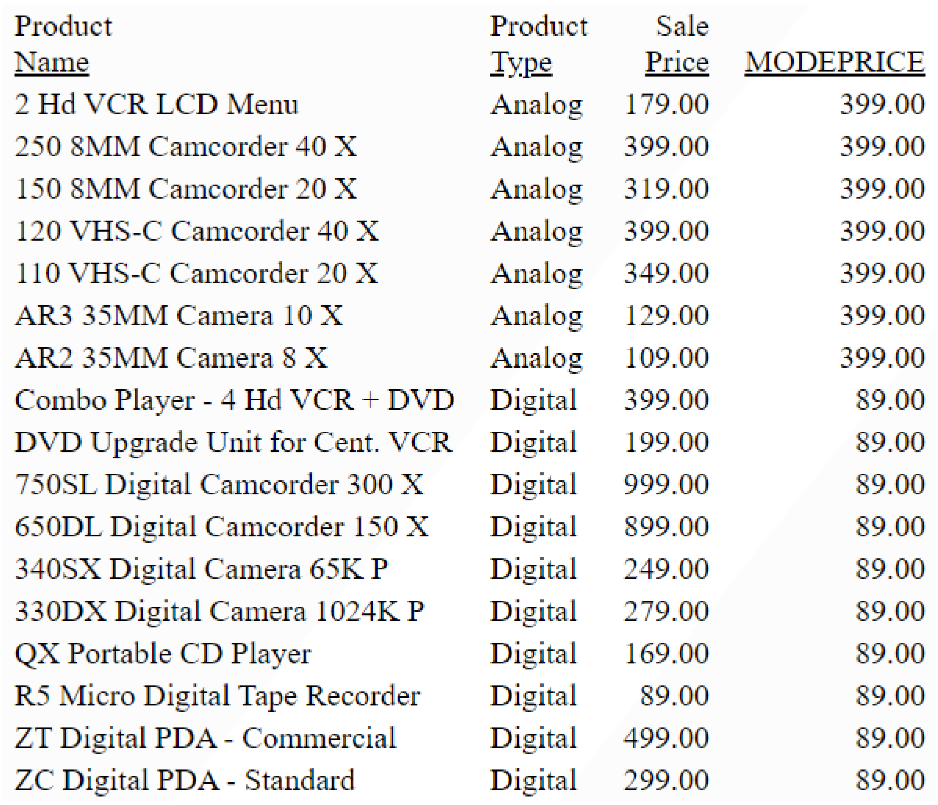


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request calculates the mode price within product category, ordered by the product name, using a window that includes the current row and one row preceding and following.

```
SQL
SELECT
   PRODNAME,
   PRODTYPE,
   PRICE,
   MODE(PRICE) OVER(PARTITION BY PRODTYPE) AS MODEPRICE
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Type	Sale Price	MODEPRICE
2 Hd VCR LCD Menu	Analog	179.00	399.00
250 8MM Camcorder 40 X	Analog	399.00	399.00
150 8MM Camcorder 20 X	Analog	319.00	399.00
120 VHS-C Camcorder 40 X	Analog	399.00	399.00
110 VHS-C Camcorder 20 X	Analog	349.00	399.00
AR3 35MM Camera 10 X	Analog	129.00	399.00
AR2 35MM Camera 8 X	Analog	109.00	399.00
Combo Player - 4 Hd VCR + DVD	Digital	399.00	89.00
DVD Upgrade Unit for Cent. VCR	Digital	199.00	89.00
750SL Digital Camcorder 300 X	Digital	999.00	89.00
650DL Digital Camcorder 150 X	Digital	899.00	89.00
340SX Digital Camera 65K P	Digital	249.00	89.00
330DX Digital Camera 1024K P	Digital	279.00	89.00
QX Portable CD Player	Digital	169.00	89.00
R5 Micro Digital Tape Recorder	Digital	89.00	89.00
ZT Digital PDA - Commercial	Digital	499.00	89.00
ZC Digital PDA - Standard	Digital	299.00	89.00

The first value of MODEPRICE (399) is the mode of the sale prices for Analog. The value 399 appears twice, while every other value appears only once.

For Digital, no value appears more than once, so the smallest value is used (89).

PERCENT_RANK: Calculating the Relative Rank of Each Row

PERCENT_RANK calculated the percentile rank of each row within a partition. For each partition, the percentile rank is zero (0) for the first row in the partition, and 100 for the last row, assuming there are multiple rows within the partition. The PARTITION BY clause is optional, but the ORDER BY clause is required. If a partition is defined, the percent rank restarts at 0 when the partition changes. The sliding window clause is not supported for PERCENT_RANK.

Syntax: How to Assign Percent Rank Numbers

```
PERCENT_RANK() OVER([PARTITION BY part1[, part2 ...]]
     ORDER BY exp1[, exp2 ...] )
```

where:

part1, part2, ...

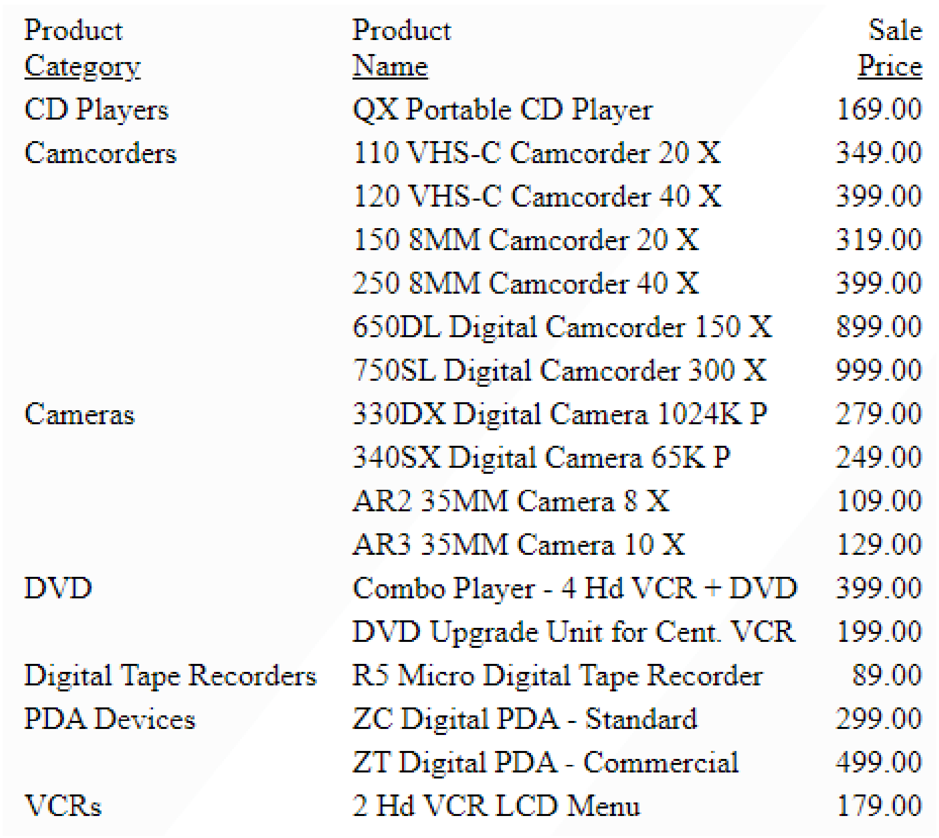
Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as ranks are assigned based on row order.

Example: Assigning Percent Rank Numbers

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

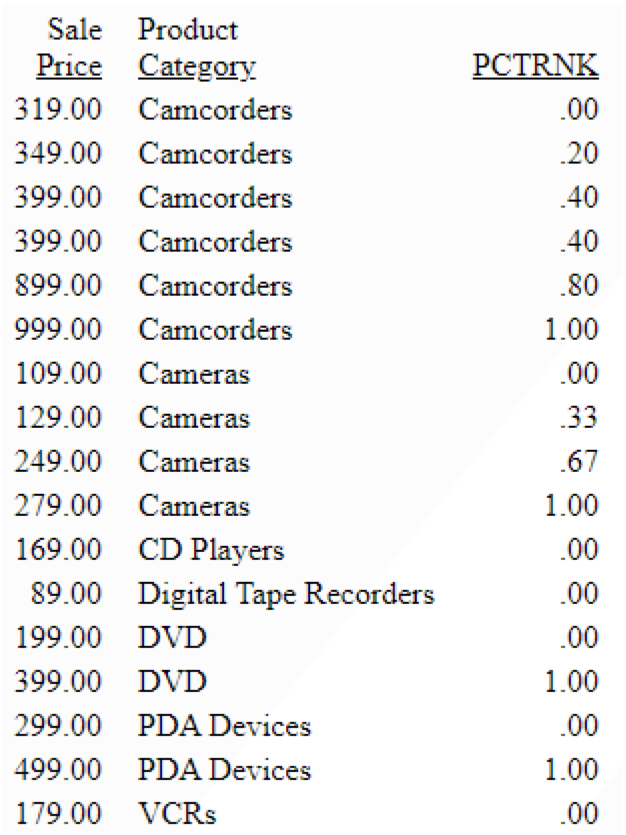


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request assigns percent rank numbers to the rows within each product category in order of price.

```
SQL
SELECT
   PRICE,
   PRODCAT,
  PERCENT_RANK()
  OVER(PARTITION BY PRODCAT  ORDER BY  PRICE) AS PCTRNK
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Sale Price	Product Category	PCTRNK
319.00	Camcorders	.00
349.00	Camcorders	.20
399.00	Camcorders	.40
399.00	Camcorders	.40
899.00	Camcorders	.80
999.00	Camcorders	1.00
109.00	Cameras	.00
129.00	Cameras	.33
249.00	Cameras	.67
279.00	Cameras	1.00
169.00	CD Players	.00
89.00	Digital Tape Recorders	.00
199.00	DVD	.00
399.00	DVD	1.00
299.00	PDA Devices	.00
499.00	PDA Devices	1.00
179.00	VCRs	.00

Within each product category, the percent ranks start at zero and, if there is more than one row in the category, end at 100.

RANK: Assigning Rank Numbers With Gaps

RANK assigns rank numbers to rows, with gaps to account for rows assigned the same rank number. The PARTITION BY clause is optional, but the ORDER BY clause is required. If a partition is defined, the rank number restarts at 1 when the partition changes. The sliding window clause is not supported for RANK.

Syntax: How to Assign Rank Numbers With Gaps

```
RANK() OVER([PARTITION BY part1[, part2 ...]]
     ORDER BY exp1[, exp2 ...] )
```

where:

part1, part2, ...

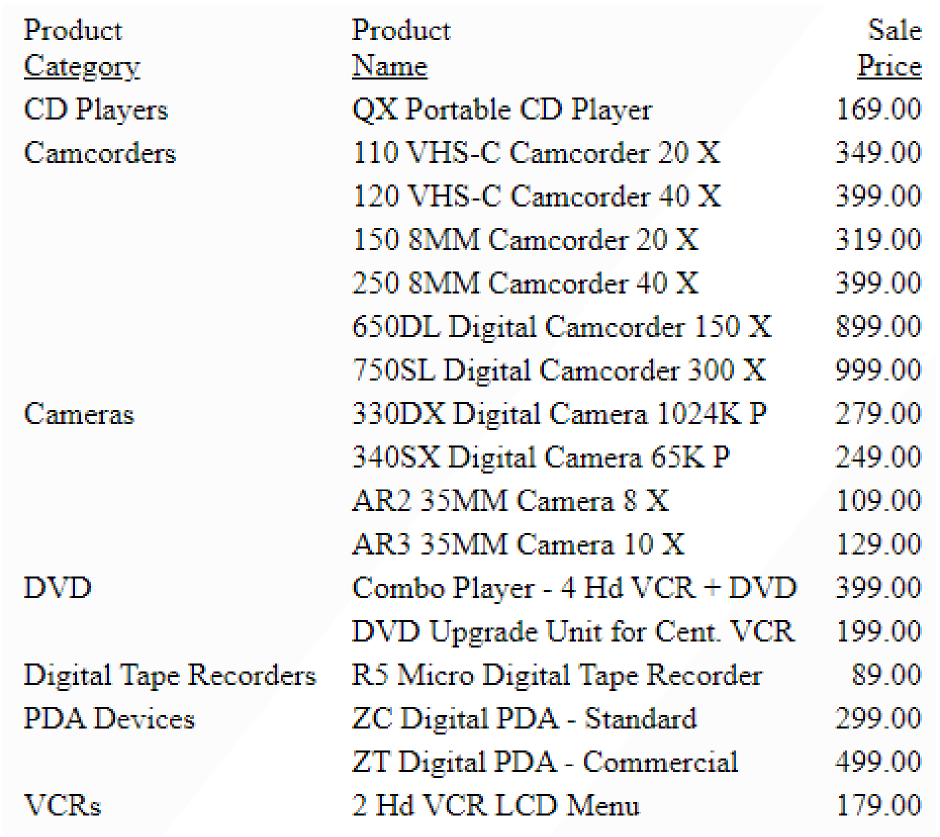
Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as ranks are assigned based on row order.

Example: Assigning Rank Numbers With No Gaps

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

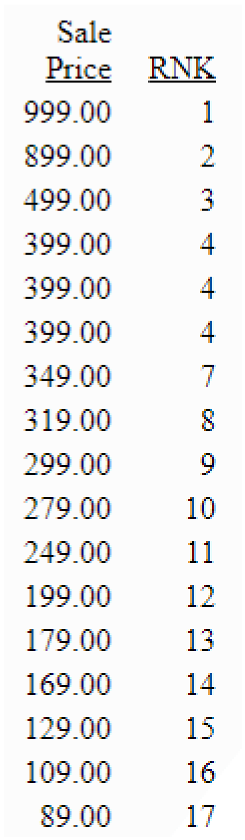


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request assigns rank numbers to all rows in decreasing order of price.

```
SQL
SELECT
  PRICE,
   RANK() OVER( ORDER BY PRICE DESC )  AS RNK ,
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Sale Price	RNK
999.00	1
899.00	2
499.00	3
399.00	4
399.00	4
399.00	4
349.00	7
319.00	8
299.00	9
279.00	10
249.00	11
199.00	12
179.00	13
169.00	14
129.00	15
109.00	16
89.00	17

Three rows have the price 399.00. Each of those rows is assigned the rank 4, and the next row is assigned the rank 7 (with the DENSE_RANK function, the ranks would go from 4 to 5).

STDDEV_POP: Calculating Population Standard Deviation Over a Group of Rows

STDDEV_POP calculates the standard deviation of a population within a partition.

Syntax: How to Calculate the Population Standard Deviation Over a Group of Rows

```
STDDEV_POP(exp) OVER([PARTITION BY part1[, part2 ...]]
     [ORDER BY exp1[, exp2 ...]] [window_frame_clause])
```

where:

exp

Is the numeric expression used in the calculation.

part1, part2 ...

Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as it changes the rows that are included in the sliding window on which the calculation is performed.

window_frame_clause

Defines the sliding window within each partition (starting row and ending row for the window). The window frame clause defines a frame around the current row within a partition, over which the analytic function is evaluated. Both physical window frames (defined by ROWS) and logical window frames (defined by RANGE) are allowed. It is your responsibility to know the syntax for your environment.

Basic syntax for the window frame clause follows:

```
{ROWS|RANGE}
{
  {UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}   |
  {BETWEEN boundary_start AND boundary_end}
}
```

The basic syntax for the start of the boundary is:

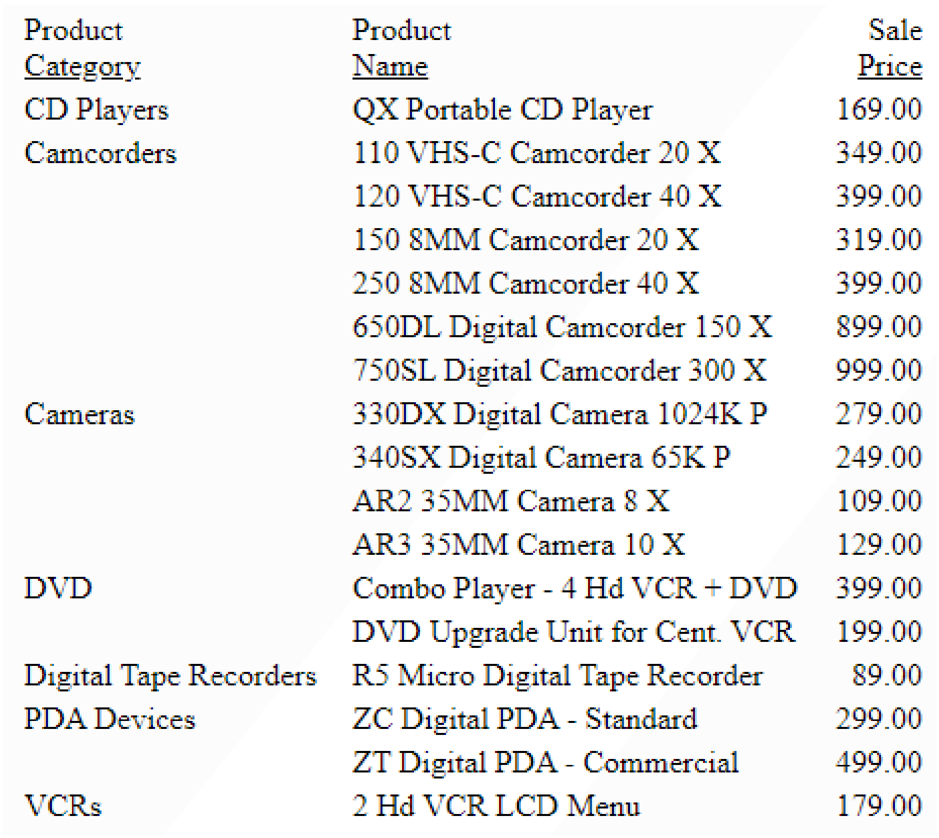
```
{UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}
```

The basic syntax for the end of the boundary is:

```
{UNBOUNDED FOLLOWING|numeric_expression {PRECEDING|FOLLOWING}|
      CURRENT ROW}
```

Example: Calculating a Population Standard Deviation Within Product Category

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

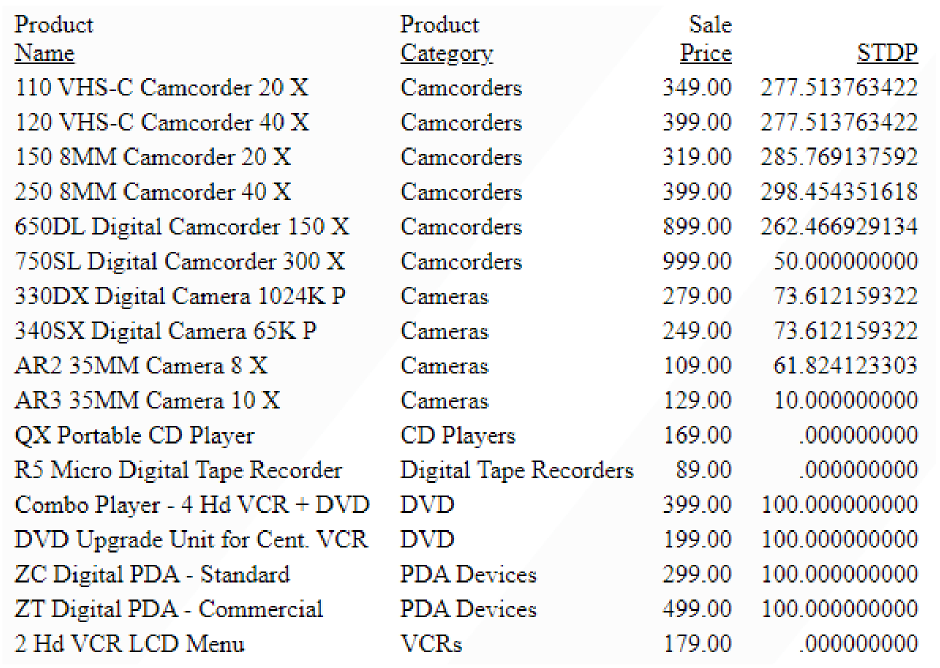


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request calculates the population standard deviation within product category, ordered by the product name, using a window that includes the current row, one row preceding, and five rows following.

```
SQL
SELECT
   PRODNAME,
   PRODCAT,
   PRICE,
   STDDEV_POP(PRICE ) OVER(PARTITION BY   PRODCAT ORDER BY  PRODNAME
     ROWS BETWEEN 1 PRECEDING AND 5 FOLLOWING)  AS STDP
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Category	Sale Price	STDP
110 VHS-C Camcorder 20 X	Camcorders	349.00	277.513763422
120 VHS-C Camcorder 40 X	Camcorders	399.00	277.513763422
150 8MM Camcorder 20 X	Camcorders	319.00	285.769137592
250 8MM Camcorder 40 X	Camcorders	399.00	298.454351618
650DL Digital Camcorder 150 X	Camcorders	899.00	262.466929134
750SL Digital Camcorder 300 X	Camcorders	999.00	50.000000000
330DX Digital Camera 1024K P	Cameras	279.00	73.612159322
340SX Digital Camera 65K P	Cameras	249.00	73.612159322
AR2 35MM Camera 8 X	Cameras	109.00	61.824123303
AR3 35MM Camera 10 X	Cameras	129.00	10.000000000
QX Portable CD Player	CD Players	169.00	.000000000
R5 Micro Digital Tape Recorder	Digital Tape Recorders	89.00	.000000000
Combo Player - 4 Hd VCR + DVD	DVD	399.00	100.000000000
DVD Upgrade Unit for Cent. VCR	DVD	199.00	100.000000000
ZC Digital PDA - Standard	PDA Devices	299.00	100.000000000
ZT Digital PDA - Commercial	PDA Devices	499.00	100.000000000
2 Hd VCR LCD Menu	VCRs	179.00	.000000000

The first value of STDP is the standard deviation of the entire camcorder category, as there is no prior row, so the window includes the current row and the following five rows.

The second value of STDP is the standard deviation of the entire camcorder category, as the window includes the prior row and only four following rows, when the category changes.

The rest of the values within the camcorder category have standard deviations based on fewer and fewer rows.

Any partition with only one row has a standard deviation of zero, as only the current row is within the window.

Any partition with two rows has a standard deviation of 100.

STDDEV_SAMP: Calculating Sample Standard Deviation Over a Group of Rows

STDDEV_SAMP calculates the standard deviation of a sample within a partition.

Syntax: How to Calculate the Sample Standard Deviation Over a Group of Rows

```
STDDEV_SAMP(exp) OVER([PARTITION BY part1[, part2 ...]]
     [ORDER BY exp1[, exp2 ...]] [window_frame_clause])
```

where:

exp

Is the numeric expression used in the calculation.

part1, part2 ...

Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as it changes the rows that are included in the sliding window on which the calculation is performed.

window_frame_clause

Defines the sliding window within each partition (starting row and ending row for the window). The window frame clause defines a frame around the current row within a partition, over which the analytic function is evaluated. Both physical window frames (defined by ROWS) and logical window frames (defined by RANGE) are allowed. It is your responsibility to know the syntax for your environment.

Basic syntax for the window frame clause follows:

```
{ROWS|RANGE}
{
  {UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}   |
  {BETWEEN boundary_start AND boundary_end}
}
```

The basic syntax for the start of the boundary is:

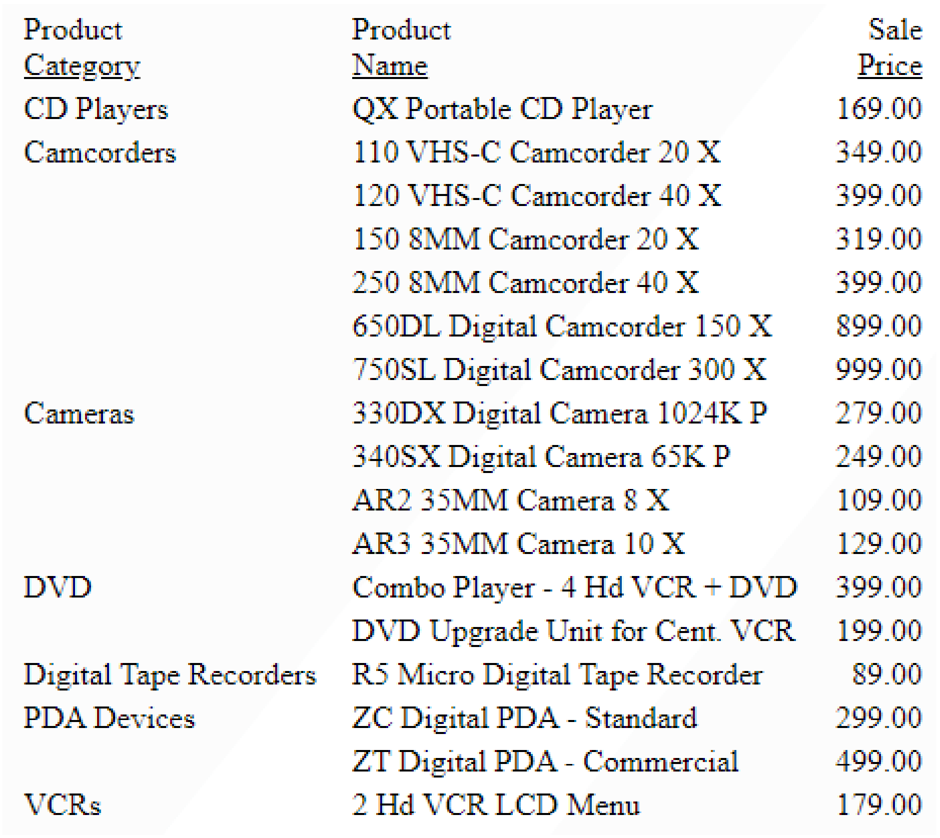
```
{UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}
```

The basic syntax for the end of the boundary is:

```
{UNBOUNDED FOLLOWING|numeric_expression {PRECEDING|FOLLOWING}|
      CURRENT ROW}
```

Example: Calculating a Sample Standard Deviation Within Product Category

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

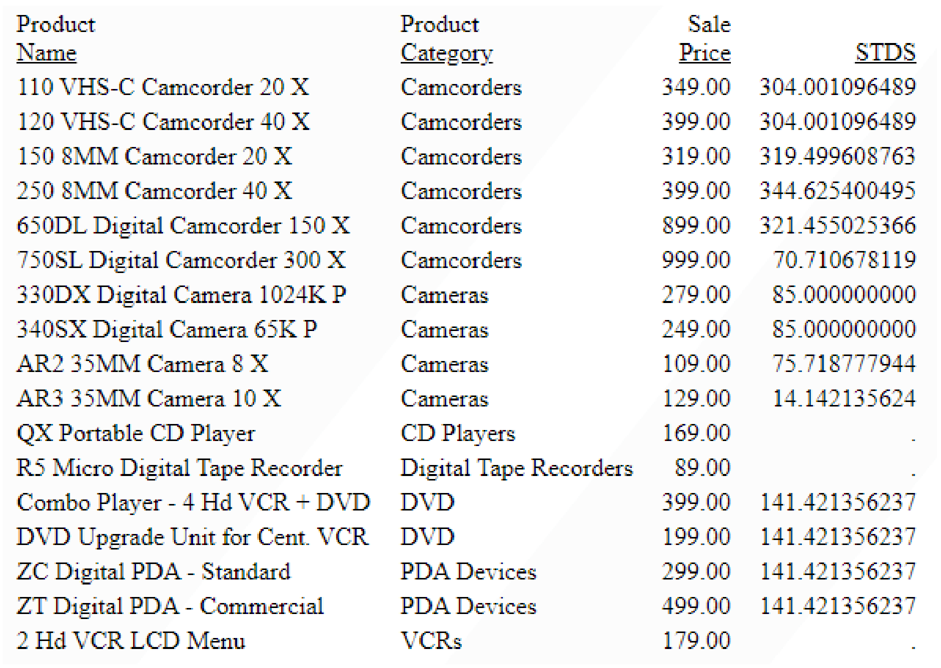


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request calculates the sample standard deviation within product category, ordered by the product name, using a window that includes the current row, one row preceding, and five rows following.

```
SQL
SELECT
   PRODNAME,
   PRODCAT,
   PRICE,
   STDDEV_SAMP(PRICE ) OVER(PARTITION BY   PRODCAT ORDER BY  PRODNAME
     ROWS BETWEEN 1 PRECEDING AND 5 FOLLOWING)  AS STDS
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Category	Sale Price	STDS
110 VHS-C Camcorder 20 X	Camcorders	349.00	304.001096489
120 VHS-C Camcorder 40 X	Camcorders	399.00	304.001096489
150 8MM Camcorder 20 X	Camcorders	319.00	319.499608763
250 8MM Camcorder 40 X	Camcorders	399.00	344.625400495
650DL Digital Camcorder 150 X	Camcorders	899.00	321.455025366
750SL Digital Camcorder 300 X	Camcorders	999.00	70.710678119
330DX Digital Camera 1024K P	Cameras	279.00	85.000000000
340SX Digital Camera 65K P	Cameras	249.00	85.000000000
AR2 35MM Camera 8 X	Cameras	109.00	75.718777944
AR3 35MM Camera 10 X	Cameras	129.00	14.142135624
QX Portable CD Player	CD Players	169.00	.
R5 Micro Digital Tape Recorder	Digital Tape Recorders	89.00	.
Combo Player - 4 Hd VCR + DVD	DVD	399.00	141.421356237
DVD Upgrade Unit for Cent. VCR	DVD	199.00	141.421356237
ZC Digital PDA - Standard	PDA Devices	299.00	141.421356237
ZT Digital PDA - Commercial	PDA Devices	499.00	141.421356237
2 Hd VCR LCD Menu	VCRs	179.00	.

The first value of STDP is the standard deviation of the entire camcorder category, as there is no prior row, so the window includes the current row and the following five rows.

The second value of STDP is the standard deviation of the entire camcorder category, as the window includes the prior row and only four following rows, when the category changes.

The rest of the values within the camcorder category have standard deviations based on fewer and fewer rows.

Any partition with only one row has a missing standard deviation, as only the current row is within the window and a sample standard deviation is based on the number of rows in the window minus one.

SUM: Summing Values Over a Group of Rows

SUM adds column values within a partition.

Syntax: How to Sum Values Over a Group of Rows

```
SUM(exp) OVER([PARTITION BY part1[, part2 ...]]
     [ORDER BY exp1[, exp2 ...]] [window_frame_clause])
```

where:

exp

Is the numeric expression used in the sum.

part1, part2 ...

Are partitioning columns or expressions.

ORDER BY *exp1, exp2 ...*

Specifies the row order within each partition. The sort order can affect the result, as it changes the rows that are included in the sliding window on which the calculation is performed.

window_frame_clause

Defines the sliding window within each partition (starting row and ending row for the window). The window frame clause defines a frame around the current row within a partition, over which the analytic function is evaluated. Both physical window frames (defined by ROWS) and logical window frames (defined by RANGE) are allowed. It is your responsibility to know the syntax for your environment.

Basic syntax for the window frame clause follows:

```
{ROWS|RANGE}
{
  {UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}  |
  {BETWEEN boundary_start AND boundary_end}
}
```

The basic syntax for the start of the boundary is:

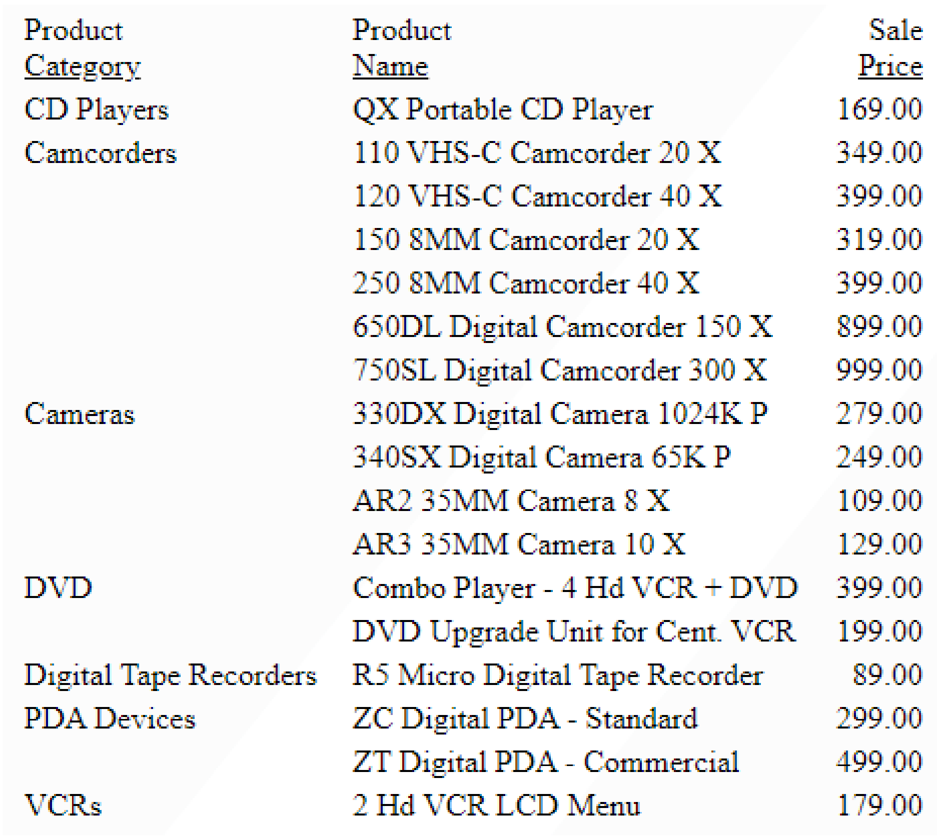
```
{UNBOUNDED PRECEDING|numeric_expression PRECEDING|CURRENT ROW}
```

The basic syntax for the end of the boundary is:

```
{UNBOUNDED FOLLOWING|numeric_expression {PRECEDING|FOLLOWING}|
      CURRENT ROW}
```

Example: Calculating a Sum Within Product Category

The dminv table created by the *DataMigrator - General* tutorial in the Server Console contains the following values:

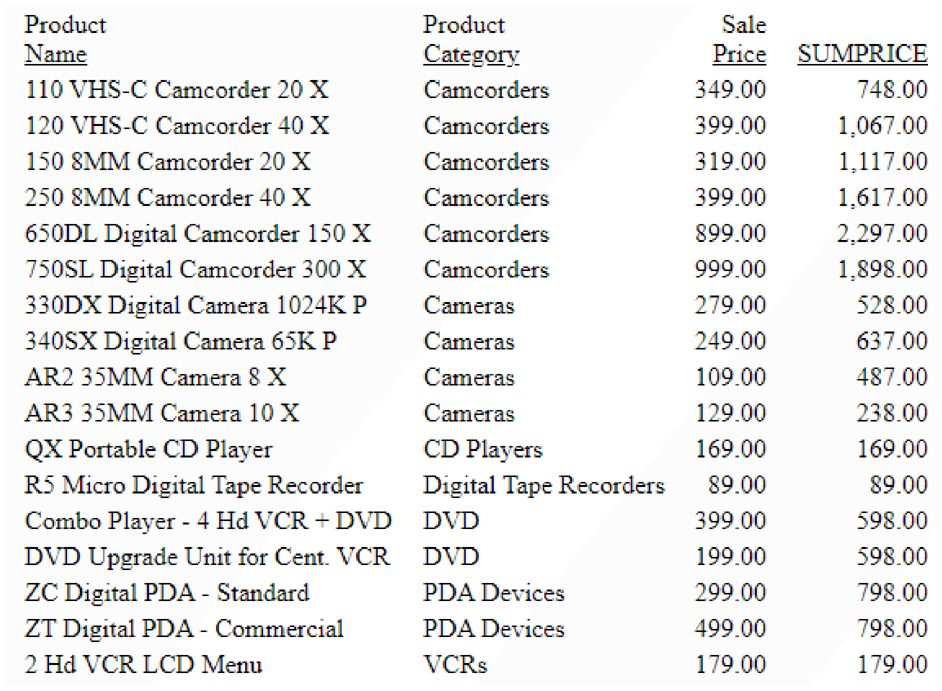


Product Category	Product Name	Sale Price
CD Players	QX Portable CD Player	169.00
Camcorders	110 VHS-C Camcorder 20 X	349.00
	120 VHS-C Camcorder 40 X	399.00
	150 8MM Camcorder 20 X	319.00
	250 8MM Camcorder 40 X	399.00
	650DL Digital Camcorder 150 X	899.00
	750SL Digital Camcorder 300 X	999.00
Cameras	330DX Digital Camera 1024K P	279.00
	340SX Digital Camera 65K P	249.00
	AR2 35MM Camera 8 X	109.00
	AR3 35MM Camera 10 X	129.00
DVD	Combo Player - 4 Hd VCR + DVD	399.00
	DVD Upgrade Unit for Cent. VCR	199.00
Digital Tape Recorders	R5 Micro Digital Tape Recorder	89.00
PDA Devices	ZC Digital PDA - Standard	299.00
	ZT Digital PDA - Commercial	499.00
VCRs	2 Hd VCR LCD Menu	179.00

The following SQL request calculates the sum price within product category, ordered by the product name, using a window that includes the current row and one row preceding and following.

```
SQL
SELECT
   PRODNAME,
   PRODCAT,
   PRICE,
   SUM(PRICE) OVER(PARTITION BY PRODCAT
      ORDER BY PRODNAME
      ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)  AS SUMPRICE
FROM
   DMINV
;
TABLE
ON TABLE SET PAGE NOLEAD
ON TABLE SET STYLE *
GRID=OFF,$
ENDSTYLE
END
```

The output is shown in the following image:



Product Name	Product Category	Sale Price	SUMPRICE
110 VHS-C Camcorder 20 X	Camcorders	349.00	748.00
120 VHS-C Camcorder 40 X	Camcorders	399.00	1,067.00
150 8MM Camcorder 20 X	Camcorders	319.00	1,117.00
250 8MM Camcorder 40 X	Camcorders	399.00	1,617.00
650DL Digital Camcorder 150 X	Camcorders	899.00	2,297.00
750SL Digital Camcorder 300 X	Camcorders	999.00	1,898.00
330DX Digital Camera 1024K P	Cameras	279.00	528.00
340SX Digital Camera 65K P	Cameras	249.00	637.00
AR2 35MM Camera 8 X	Cameras	109.00	487.00
AR3 35MM Camera 10 X	Cameras	129.00	238.00
QX Portable CD Player	CD Players	169.00	169.00
R5 Micro Digital Tape Recorder	Digital Tape Recorders	89.00	89.00
Combo Player - 4 Hd VCR + DVD	DVD	399.00	598.00
DVD Upgrade Unit for Cent. VCR	DVD	199.00	598.00
ZC Digital PDA - Standard	PDA Devices	299.00	798.00
ZT Digital PDA - Commercial	PDA Devices	499.00	798.00
2 Hd VCR LCD Menu	VCRs	179.00	179.00

The first value of SUMPRICE (748.00) is the sum of the first two sale price values (current row, 349, and following row, 399), as there is no preceding row in the partition.

The second value of SUMPRICE (1967.00) is the sum of the sale prices in rows 1, 2, and 3 (349, 399, and 319).

This continues until the end of the partition (last Camcorders row), when the SUMPRICE value (1898.00) is the sum of the sale price in that row (999) and the preceding row (899).

For partitions that consist of one row, such as Digital Tape Recorders, the SUMPRICE value is the same as the sale price value, as there is no preceding or following row.

Chapter 27

SQL Statistical Functions

SQL statistical functions calculate common statistical measures.

In this chapter:

CORRELATION: Calculating the Degree of Correlation Between Two Sets of Data

The CORRELATION function calculates the correlation coefficient between two numeric fields. The function returns a numeric value between zero (-1.0) and 1.0.

Syntax: How to Calculate the Correlation Coefficient Between Two Fields

```
CORRELATION(field1, field2)
```

where:

field1

Numeric

Is the first set of data for the correlation.

field2

Numeric

Is the second set of data for the correlation.

Note: Arguments for CORRELATION cannot be prefixed fields. If you need to work with fields that have a prefix operator applied, apply the prefix operators to the fields in COMPUTE commands and save the results in a HOLD file. Then, run the correlation against the HOLD file.

Example: Calculating a Correlation

CORRELATION calculates the correlation between DOLLARS and BUDDOLLARS.

```
CORRELATION(DOLLARS, BUDDOLLARS)
```

For DOLLARS=46,156,290.00 and BUDDOLLARS=46,220,778.00, the result is 0.895691073.

STDDEV_POP: Calculating the Standard Deviation of an Entire Population

The standard deviation is the square root of the variance, which is a measure of how numeric observations deviate from their expected value (mean). STDDEV_POP returns a numeric value that represents the amount of dispersion in the entire population. Therefore, the divisor in the standard deviation calculation (also called degrees of freedom) will be the total number of data points, N.

Syntax: How to Calculate the Standard Deviation of an Entire Population

```
STDDEV_POP(field)
```

where:

field

Numeric

Is the field containing the set of observations for the standard deviation calculation.

Example: Calculating the Standard Deviation of an Entire Population

STDDEV_POP calculates the standard deviation of the entire set of observations of dollars.

```
STDDEV_POP(DOLLARS)
```

For 46,156,290, the result is 6,156.997845651.

STDDEV_SAMP: Calculating the Standard Deviation of a Sample of a Population

The standard deviation is the square root of the variance, which is a measure of how numeric observations deviate from their expected value (mean). STDDEV_SAMP returns a numeric value that represents the amount of dispersion in a sample of the population. Therefore, the divisor in the standard deviation calculation (also called degrees of freedom) will be the total number of data points minus 1, N-1.

Syntax: How to Calculate the Standard Deviation of a Sample of the Population

```
STDDEV_SAMP(field)
```

where:

`field`

Numeric

Is the field containing the set of observations for the standard deviation calculation.

Example: Calculating the Standard Deviation of a Sample of a Population

STDDEV_POP calculates the standard deviation of a sample of the set of observations of dollars.

```
STDDEV_SAMP(DOLLARS)
```

For 46,156,290, the result is 6,157.711080272.

Legal and Third-Party Notices

Index