Informatica®
10.5.1

Transformation Language Reference

Informatica Transformation Language Reference
10.5.1
September 2021

Publication Date: 2022-07-27

Table of Contents

Preface 9

Informatica Resources 9

- Informatica Network 9
- Informatica Knowledge Base 9
- Informatica Documentation 9
- Informatica Product Availability Matrices 10
- Informatica Velocity 10
- Informatica Marketplace 10
- Informatica Global Customer Support 10

Chapter 1: The Transformation Language 11

The Transformation Language Overview 11

- Transformation Language Components 11
- Internationalization and the Transformation Language 12

Expression Syntax 12

- Expression Components 12
- Rules and Guidelines for Expression Syntax 13

Adding Comments to Expressions 14

Reserved Words 15

Chapter 2: Constants 16

DD_DELETE 16

- Example 16

DD_INSERT 16

- Examples 17

DD_REJECT 17

- Examples 17

DD_UPDATE 17

- Examples 18

FALSE 18

- Example 18

NULL 18

- Working with Null Values in Boolean Expressions 19
- Null Values in Comparison Expressions 19
- Null Values in Aggregate Functions 19
- Null Values in Filter Conditions 19
- Nulls with Operators 19

TRUE 19

- Example 20

Chapter 3: Operators. 21

Operator Precedence. 21
Complex Operators. 22
Subscript Operator. 23
Dot Operator. 24
Complex Operators for Nested Data Types. 26
Arithmetic Operators. 30
String Operators. 31
Nulls. 31
Example. 31
Comparison Operators. 31
Logical Operators. 33
Nulls. 33

Chapter 4: Variables. 34

Built-in Variables. 34
SYSDATE. 34
Local Variables. 34

Chapter 5: Dates. 35

Dates Overview. 35
Date/Time Datatype. 35
Julian Day, Modified Julian Day, and the Gregorian Calendar. 36
Dates in the Year 2000. 36
Dates in Relational Databases. 38
Dates in Flat Files. 38
Default Date Format. 38
Date Format Strings. 39
TO_CHAR Format Strings. 40
Examples. 42
TO_DATE and IS_DATE Format Strings. 43
Rules and Guidelines for Date Format Strings. 45
Example. 45
Understanding Date Arithmetic. 47

Chapter 6: Functions. 48

Function Categories. 48
Aggregate Functions. 48
Aggregate Functions and Nulls. 50
Character Functions. 50
Complex Functions. 51
Conversion Functions. 52

Data Cleansing Functions. . . . 52
Date Functions. . . . 53
Encoding Functions. . . . 53
Financial Functions. . . . 54
Numeric Functions. . . . 54
Scientific Functions. . . . 54
Special Functions. . . . 55
String Functions. . . . 55
Test Functions. . . . 55
Window Functions. . . . 55
ABORT. . . . 56
ABS. . . . 56
ADD_TO_DATE. . . . 57
AES_DECRYPT. . . . 60
AES_ENCRYPT. . . . 61
ANY. . . . 62
ARRAY. . . . 63
ASCII. . . . 64
AVG. . . . 65
CAST. . . . 66
CEIL. . . . 67
CHOOSE. . . . 68
CHR. . . . 69
CHRCODE. . . . 70
COLLECT_LIST. . . . 71
COLLECT_MAP. . . . 71
COMPRESS. . . . 72
CONCAT. . . . 73
CONCAT_ARRAY. . . . 75
CONVERT_BASE. . . . 75
COS. . . . 76
COSH. . . . 77
COUNT. . . . 78
CRC32. . . . 80
CREATE_TIMESTAMP_TZ. . . . 81
CUME. . . . 82
DATE_COMPARE. . . . 83
DATE_DIFF. . . . 84
DEC_BASE64. . . . 87
DECODE. . . . 88
DECOMPRESS. . . . 90
ENC_BASE64. . . . 90

ERROR. 91
EXP. 92
EXTRACT_STRUCT. 93
FIRST. 93
FLOOR. 95
FV 96
GET_DATE_PART. 97
GET_TIMEZONE. 99
GET_TIMESTAMP. 99
GREATEST. 100
IIF. 101
IN. 104
INDEXOF. 105
INITCAP. 106
INSTR. 107
ISNULL. 110
IS_DATE. 111
IS_NUMBER. 113
IS_SPACES. 115
LAG. 116
LAST. 118
LAST_DAY. 119
LEAD. 120
LEAST. 122
LENGTH. 123
LN. 124
LOG. 125
LOWER. 125
LPAD. 126
LTRIM. 128
MAKE_DATE_TIME. 129
MAP. 130
MAP_FROM_ARRAYS. 131
MAP_KEYS. 132
MAP_VALUES. 133
MAX (Dates). 134
MAX (Numbers). 135
MAX (String). 136
MD5. 137
MEDIAN. 138
METAPHONE. 140
MIN (Dates). 143

MIN (Numbers). 144
MIN (String). 146
MOD. 147
MOVINGAVG. 148
MOVINGSUM. 150
NPER. 151
PARSE_JSON. 152
PARSE_XML. 153
PERCENTILE. 154
PMT. 156
POWER. 157
PV. 158
RAND. 159
RATE. 160
REG_EXTRACT. 160
REG_MATCH. 163
REG_REPLACE. 164
REPLACECHR. 165
REPLACESTR. 168
RESPEC. 171
REVERSE. 172
ROUND (Dates). 173
ROUND (Numbers). 177
RPAD. 179
RTRIM. 180
SET_DATE_PART. 182
SIGN. 184
SIN. 185
SINH. 186
SIZE. 187
SOUNDEX. 188
SQL_LIKE. 190
SQRT. 191
STDDEV. 192
STRUCT. 193
STRUCT_AS. 195
SUBSTR. 196
SUM. 198
SYSTIMESTAMP. 199
TAN. 200
TANH. 201
TIME_RANGE. 202

TO_BIGINT. 203
TO_CHAR (Dates). 205
TO_CHAR (Numbers). 209
TO_DATE. 211
TO_DECIMAL. 214
TO_DECIMAL38. 215
TO_FLOAT. 216
TO_INTEGER. 217
TO_TIMESTAMP_TZ. 219
TRUNC (Dates). 220
TRUNC (Numbers). 223
UPPER. 225
UUID4. 226
UUID_UNPARSE. 226
VARIANCE. 226

Index. 228

Preface

Refer to the *Informatica® Developer Transformation Language Reference* to understand transformation language in the Developer tool. Learn how you can use constants, operators, variables, dates, and functions to transform source data.

Informatica Resources

Informatica provides you with a range of product resources through the Informatica Network and other online portals. Use the resources to get the most from your Informatica products and solutions and to learn from other Informatica users and subject matter experts.

Informatica Network

The Informatica Network is the gateway to many resources, including the Informatica Knowledge Base and Informatica Global Customer Support. To enter the Informatica Network, visit https://network.informatica.com.

As an Informatica Network member, you have the following options:

- Search the Knowledge Base for product resources.
- View product availability information.
- Create and review your support cases.
- Find your local Informatica User Group Network and collaborate with your peers.

Informatica Knowledge Base

Use the Informatica Knowledge Base to find product resources such as how-to articles, best practices, video tutorials, and answers to frequently asked questions.

To search the Knowledge Base, visit https://search.informatica.com. If you have questions, comments, or ideas about the Knowledge Base, contact the Informatica Knowledge Base team at KB_Feedback@informatica.com.

Informatica Documentation

Use the Informatica Documentation Portal to explore an extensive library of documentation for current and recent product releases. To explore the Documentation Portal, visit https://docs.informatica.com.

If you have questions, comments, or ideas about the product documentation, contact the Informatica Documentation team at infa_documentation@informatica.com.

Informatica Product Availability Matrices

Product Availability Matrices (PAMs) indicate the versions of the operating systems, databases, and types of data sources and targets that a product release supports. You can browse the Informatica PAMs at https://network.informatica.com/community/informatica-network/product-availability-matrices.

Informatica Velocity

Informatica Velocity is a collection of tips and best practices developed by Informatica Professional Services and based on real-world experiences from hundreds of data management projects. Informatica Velocity represents the collective knowledge of Informatica consultants who work with organizations around the world to plan, develop, deploy, and maintain successful data management solutions.

You can find Informatica Velocity resources at http://velocity.informatica.com. If you have questions, comments, or ideas about Informatica Velocity, contact Informatica Professional Services at ips@informatica.com.

Informatica Marketplace

The Informatica Marketplace is a forum where you can find solutions that extend and enhance your Informatica implementations. Leverage any of the hundreds of solutions from Informatica developers and partners on the Marketplace to improve your productivity and speed up time to implementation on your projects. You can find the Informatica Marketplace at https://marketplace.informatica.com.

Informatica Global Customer Support

You can contact a Global Support Center by telephone or through the Informatica Network.

To find your local Informatica Global Customer Support telephone number, visit the Informatica website at the following link:
https://www.informatica.com/services-and-training/customer-success-services/contact-us.html.

To find online support resources on the Informatica Network, visit https://network.informatica.com and select the eSupport option.

CHAPTER 1

The Transformation Language

This chapter includes the following topics:

- The Transformation Language Overview, 11
- Expression Syntax, 12
- Adding Comments to Expressions, 14
- Reserved Words, 15

The Transformation Language Overview

Informatica Developer provides a transformation language that includes SQL-like functions to transform source data. Use these functions to write expressions.

Expressions modify data or test whether data matches conditions. For example, you might use the AVG function to calculate the average salary of all the employees, or the SUM function to calculate the total sales for a specific branch.

You can create a simple expression that only contains a port, such as ORDERS, or a numeric literal, such as 10. You can also write complex expressions that include functions nested within functions, or combine different ports using the transformation language operators.

Transformation Language Components

The transformation language includes the following components to create simple or complex transformation expressions:

- **Functions.** Over 100 SQL-like functions allow you to change data in a mapping.
- **Operators.** Use transformation operators to create transformation expressions to perform mathematical computations, combine data, or compare data.
- **Constants.** Use built-in constants to reference values that remain constant, such as TRUE.
- **Mapping parameters.** Create mapping parameters for use within a mapping or mapplet to reference values that remain constant throughout a mapping or mapplet run, such as a state sales tax rate.
- **Built-in and local variables.** Use built-in variables to write expressions that reference values that vary, such as the system date. You can also create local variables in transformations.
- **Return values.** You can also write expressions that include the return values Lookup transformations.

Internationalization and the Transformation Language

Transformation language functions can handle character data in either ASCII or Unicode data movement mode. Use Unicode mode to handle *multibyte* character data. The return values of the following functions and transformations depend on the code page of the Data Integration Service and the data movement mode:

- INITCAP
- LOWER
- UPPER
- MIN (Date)
- MIN (Number)
- MIN (String)
- MAX (Date)
- MAX (Number)
- MAX (String)
- Any function that uses conditional statements to compare strings, such as IIF and DECODE

MIN and MAX also return values based on the sort order associated with the Data Integration Service code page.

When you validate an invalid expression in the Expression Editor, a dialog box displays the expression with an error indicator, ">>>>". This indicator appears to the left of and points to the part of the expression containing the error. For example, if the expression a = b + c contains an error at c, the error message displays:

```
a = b +   >>>> c
```

Transformation language functions that evaluate character data are character-oriented, not byte-oriented. For example, the LENGTH function returns the number of characters in a string, not the number of bytes. The LOWER function returns a string in lowercase based on the code page of the Data Integration Service.

Expression Syntax

Although the transformation language is based on standard SQL, there are difference between the two languages. For example, SQL supports the keywords ALL and DISTINCT for aggregate functions, but the transformation language does not. On the other hand, the transformation language supports an optional filter condition for aggregate functions, while SQL does not.

You can create an expression that is as simple as a port (such as ORDERS) or a numeric literal (such as 10). You can also write complex expressions that include functions nested within functions, or combine different columns using the transformation language operators.

Expression Components

Expressions can consist of any combination of the following components:

- Ports (input, input/output, variable)
- String literals, numeric literals
- Constants
- Functions

- Built-in and local variables
- Mapping parameters
- Operators
- Return values

Ports and Return Values

When you write an expression that includes a port or return value from an unconnected transformation, use the reference qualifiers in the following table:

Reference Qualifier	Description
:LKP	Required when you create an expression that includes the return value from an unconnected Lookup transformation. The general syntax is: `:LKP.lookup_transformation(argument1, argument2, ...)` The arguments are the local ports used in the lookup condition. The order must match the order of the ports in the transformation. The datatypes for the local ports must match the datatype of the Lookup ports used in the lookup condition.

String and Numeric Literals

You can include numeric or string literals.

Be sure to enclose string literals within single quotation marks. For example:

```
'Alice Davis'
```

String literals are case sensitive and can contain any character except a single quotation mark. For example, the following string is not allowed:

```
'Joan's car'
```

To return a string containing a single quote, use the CHR function:

```
'Joan' || CHR(39) || 's car'
```

Do not use single quotation marks with numeric literals. Just enter the number you want to include. For example:

```
.05
```

or

```
$$Sales_Tax
```

Rules and Guidelines for Expression Syntax

Use the following rules and guidelines when you write expressions:

- You cannot include both single-level and nested aggregate functions in an Aggregator transformation.
- If you need to create both single-level and nested functions, create separate Aggregator transformations.
- You cannot use strings in numeric expressions.

  For example, the expression `1 + '1'` is not valid because you can only perform addition on numeric datatypes. You cannot add an integer and a string.
- You cannot use strings as numeric parameters.

For example, the expression `SUBSTR(TEXT_VAL, '1', 10)` is not valid because the SUBSTR function requires an integer value, not a string, as the start position.

- You cannot mix datatypes when using comparison operators.

  For example, the expression `123.4 = '123.4'` is not valid because it compares a decimal value with a string.
- You can pass a value from a port, literal string or number, Lookup transformation, or the results of another expression.
- Use the ports tab in the Expression Editor to enter a port name into an expression. If you rename a port in a connected transformation, the Developer tool propagates the name change to expressions in the transformation.
- Separate each argument in a function with a comma.
- Except for literals, the transformation language is not case sensitive.
- Except for literals, the Developer tool and Data Integration Service ignore spaces.
- The colon (:), comma (,), and period (.) have special meaning and should be used only to specify syntax.
- The Data Integration Service treats a dash (-) as a minus operator.
- If you pass a literal value to a function, enclose literal strings within single quotation marks. Do not use quotation marks for literal numbers. The Data Integration Service treats any string value enclosed in single quotation marks as a character string.
- When you pass a mapping parameter to a function within an expression, do not use quotation marks to designate mapping parameters.
- Do not use quotation marks to designate ports.
- You can nest multiple functions within an expression except aggregate functions, which allow only one nested aggregate function. The Data Integration Service evaluates the expression starting with the innermost function.

Adding Comments to Expressions

The transformation language provides two comment specifiers to let you insert comments in expressions:

- Two dashes, as in:

  ```
  -- These are comments
  ```
- Two slashes, as in:

  ```
  // These are comments
  ```

The Data Integration Service ignores all text on a line preceded by these two comment specifiers. For example, if you want to concatenate two strings, you can enter the following expression with comments in the middle of the expression:

```
 -- This expression concatenates first and last names for customers:
FIRST_NAME -- First names from the CUST table
|| // Concat symbol
LAST_NAME // Last names from the CUST table
// Joe Smith Aug 18 1998
```

The Data Integration Service ignores the comments and evaluates the expression as follows:

```
FIRST_NAME || LAST_NAME
```

You cannot continue a comment to a new line:

```
-- This expression concatenates first and last names for customers:
FIRST_NAME -- First names from the CUST table
|| // Concat symbol
LAST_NAME // Last names from the CUST table
Joe Smith Aug 18 1998
```

In this case, the Developer tool does not validate the expression, since the last line is not a valid expression.

If you do not want to embed comments, you can add them by clicking Comment in the Expression Editor.

Reserved Words

Some keywords in the transformation language, such as constants, operators, and built-in variables, are reserved for specific functions. These include:

- :INFA
- :LKP
- :MCR
- :TYPE
- AND
- DD_DELETE
- DD_INSERT
- DD_REJECT
- DD_UPDATE
- FALSE
- NOT
- NULL
- OR
- PROC_RESULT
- SPOUTPUT
- SYSDATE
- TRUE

Note: You cannot use a reserved word to name a port or local variable. You can only use reserved words within transformation expressions. Reserved words have predefined meanings in expressions.

CHAPTER 2

Constants

This chapter includes the following topics:

- DD_DELETE, 16
- DD_INSERT, 16
- DD_REJECT, 17
- DD_UPDATE, 17
- FALSE, 18
- NULL, 18
- TRUE, 19

DD_DELETE

Flags records for deletion in an update strategy expression. DD_DELETE is equivalent to the integer literal 2.

Note: Use the DD_DELETE constant in the Update Strategy transformation only. Use DD_DELETE instead of the integer literal 2 to facilitate troubleshooting complex numeric expressions.

Example

The following expression marks items with an ID number of 1001 for deletion, and all other items for insertion:

```
IIF( ITEM_ID = 1001, DD_DELETE, DD_INSERT )
```

This update strategy expression uses numeric literals to produce the same result:

```
IIF( ITEM_ID = 1001, 2, 0 )
```

Note: The expression using constants is easier to read than the expression using numeric literals.

DD_INSERT

Flags records for insertion in an update strategy expression. DD_INSERT is equivalent to the integer literal 0.

Note: Use the DD_INSERT constant in the Update Strategy transformation only. Use DD_INSERT instead of the integer literal 0 to facilitate troubleshooting complex numeric expressions.

Examples

The following examples modify a mapping that calculates monthly sales by salesperson, so you can examine the sales of just one salesperson.

The following update strategy expression flags an employee's sales for insertion, and rejects everything else:

```
IIF( EMPLOYEENAME = 'Alex', DD_INSERT, DD_REJECT )
```

This update strategy expression uses numeric literals to produce the same result:

```
IIF( EMPLOYEENAME = 'Alex', 0, 3 )
```

Tip: The expression using constants is easier to read than the expression using numeric literals.

DD_REJECT

Flags records for rejection in an update strategy expression. DD_REJECT is equivalent to the integer literal 3.

Note: Use the DD_REJECT constant in the Update Strategy transformation only. Use DD_REJECT instead of the integer literal 3 to facilitate troubleshooting complex numeric expressions.

Use DD_REJECT to filter or validate data. If you flag a record as reject, the Data Integration Service skips the record and writes it to the session reject file.

Examples

The following examples modify a mapping that calculates the sales for the current month, so it includes only positive values.

This update strategy expression flags records less than 0 for reject and all others for insert:

```
IIF( SALES > 0, DD_INSERT, DD_REJECT )
```

This expression uses numeric literals to produce the same result:

```
IIF( SALES > 0, 0, 3 )
```

The expression using constants is easier to read than the expression using numeric literals.

The following data-driven example uses DD_REJECT and IS_SPACES to avoid writing spaces to a character column in a target table. This expression flags records that consist entirely of spaces for reject and flags all others for insert:

```
IIF( IS_SPACES( CUST_NAMES ), DD_REJECT, DD_INSERT )
```

DD_UPDATE

Flags records for update in an update strategy expression. DD_UPDATE is equivalent to the integer literal 1.

Note: Use the DD_UPDATE constant in the Update Strategy transformation only. Use DD_UPDATE instead of the integer literal 1 to facilitate troubleshooting complex numeric expressions.

Examples

The following examples modify a mapping that calculates sales for the current month. The mapping loads sales for one employee.

This expression flags records for Alex as updates and flags all others for rejection:

```
IIF( EMPLOYEENAME = 'Alex', DD_UPDATE, DD_REJECT )
```

This expression uses numeric literals to produce the same result, flagging Alex's sales for update (1) and flagging all other sales records for rejection (3):

```
IIF( EMPLOYEENAME = 'Alex', 1, 3 )
```

The expression using constants is easier to read than the expression using numeric literals.

The following update strategy expression uses SYSDATE to find only those orders that have shipped in the last two days and flag them for insertion. Using DATE_DIFF, the expression subtracts DATE_SHIPPED from the system date, returning the difference between the two dates. Because DATE_DIFF returns a Double value, the expression uses TRUNC to truncate the difference. It then compares the result to the integer literal 2. If the result is greater than 2, the expression flags the records for rejection. If the result is 2 or less, it flags the records for update. Otherwise, it flags them for rejection:

```
IIF( TRUNC( DATE_DIFF( SYSDATE, ORDERS_DATE_SHIPPED, 'DD' ), 0 ) > 2, DD_REJECT,
DD_UPDATE )
```

FALSE

Clarifies a conditional expression. FALSE is equivalent to the integer 0.

Example

The following example uses FALSE in a DECODE expression to return values based on the results of a comparison. This is useful if you want to perform multiple searches based on a single search value:

```
DECODE( FALSE,
Var1 = 22,'Variable 1 was 22!',
Var2 = 49,'Variable 2 was 49!',
Var1 < 23, 'Variable 1 was less than 23.',
Var2 > 30, 'Variable 2 was more than 30.',
'Variables were out of desired ranges.')
```

NULL

Indicates that a value is either unknown or undefined. NULL is not equivalent to a blank or empty string (for character columns) or 0 (for numerical columns).

Although you can write expressions that return nulls, any column that has the NOT NULL or PRIMARY KEY constraint will not accept nulls. Therefore, if the Data Integration Service tries to write a null value to a column with one of these constraints, the database will reject the row and the Data Integration Service will write it to the reject file. Be sure to consider nulls when you create transformations.

Functions can handle nulls differently. If you pass a null value to a function, it might return 0 or NULL, or it might ignore null values.

Working with Null Values in Boolean Expressions

Expressions that combine a null value with a Boolean expression produces results that are ANSI compliant. For example, the Data Integration Service produces the following results:

- NULL AND TRUE = NULL
- NULL AND FALSE = FALSE

Null Values in Comparison Expressions

When you use a null value in an expression containing a comparison operator, the Data Integration Service produces a null value. To check for null values in columns, you must use the ISNULL() in comparison expressions.

To return rows that do not contain null values, use the ISNULL function instead of the constant `!=`. For example, use `NOT ISNULL(Field_A)`.

The following expression results in a null value, and the Filter transformation does not return any rows: `Field_A!=NULL`.

You can also configure the Lookup transformation to treat null values as high or low in comparison operations. Use the Null Ordering property in the lookup source to configure how the Data Integration Service handles null values in comparison expressions in the Lookup transformation.

Null Values in Aggregate Functions

The Data Integration Service treats null values as nulls in aggregate functions. If you pass an entire port or group of null values, the function returns NULL.

Null Values in Filter Conditions

If a filter condition evaluates to NULL, the function does not select the record. If the filter condition evaluates to NULL for all records in the selected port, the aggregate function returns NULL (except COUNT, which returns 0). You can use filter conditions with aggregate functions and the CUME, MOVINGAVG, and MOVINGSUM functions.

Nulls with Operators

Any expression that uses operators (except the string operator ||) and contains a null value always evaluates to NULL. For example, the following expression evaluates to NULL:

```
8 * 10 - NULL
```

To test for nulls, use the ISNULL function.

TRUE

Returns a value based on the result of a comparison. TRUE is equivalent to the integer 1.

Example

The following example uses TRUE in a DECODE expression to return values based on the results of a comparison. This is useful if you want to perform multiple searches based on a single search value:

```
DECODE( TRUE,
Var1 = 22,'Variable 1 was 22!',
Var2 = 49,'Variable 2 was 49!',
Var1 < 23, 'Variable 1 was less than 23.',
Var2 > 30, 'Variable 2 was more than 30.',
'Variables were out of desired ranges.')
```

CHAPTER 3

Operators

This chapter includes the following topics:

- Operator Precedence, 21
- Complex Operators, 22
- Arithmetic Operators, 30
- String Operators, 31
- Comparison Operators, 31
- Logical Operators, 33

Operator Precedence

The transformation language supports the use of multiple operators and the use of operators within nested expressions.

If you write an expression that includes multiple operators, the Data Integration Service evaluates the expression in the following order:

1. Complex operators
2. Arithmetic operators
3. String operators
4. Comparison operators
5. Logical operators

The Data Integration Service evaluates operators in the order they appear in the following table. It evaluates operators in an expression with equal precedence to all operators from left to right.

The following table lists the precedence for all transformation language operators:

Operator	Meaning
`[ ], .`	Subscript, dot.
( )	Parentheses.
+, -, NOT	Unary plus and minus and the logical NOT operator.
*, /,%	Multiplication, division, modulus.

Operator	Meaning
+, -	Addition, subtraction.
||	Concatenate.
<, <=, >, >=	Less than, less than or equal to, greater than, greater than or equal to.
=, <>, !=, ^=	Equal to, not equal to, not equal to, not equal to.
AND	Logical AND operator, used when specifying conditions.
OR	Logical OR operator, used when specifying conditions.

The transformation language also supports the use of operators within nested expressions. When expressions contain parentheses, the Data Integration Service evaluates operations inside parentheses before operations outside parentheses. Operations in the innermost parentheses are evaluated first.

For example, depending on how you nest the operations, the equation 8 + 5 - 2 * 8 returns different values:

Equation	Return Value
8 + 5 - 2 * 8	-3
8 + (5 - 2) * 8	32

Complex Operators

Use complex operators to access elements in a complex data type. You can access elements in an array, map, or struct data type.

You can use complex operators in mappings that run on the Spark engine.

The following table lists the complex operators in the transformation language:

Operator	Meaning
[ ]	Subscript operator. Use a subscript operator to access one or more elements in an array. You can also use a subscript operator to access the value corresponding to a given key in a key-value pair of a map.
.	Dot operator. Use a dot operator to access an element in a struct. You can also use a dot operator in an array of structs to access elements in each struct.

When you use a dot operator in an array of structs, it returns the elements of the same name within each struct as an array. To access elements in a nested array or struct, you can use a combination of complex operators.

Subscript Operator

Use a subscript operator to access elements in an array or a map. You can access a specific element or a range of elements in an array. You can access the value corresponding to a given key in a key-value pair of a map.

Syntax

To access a specific element in an array, use the following syntax:

```
array[ index ]
```

To access a range of elements in an array, use the following syntax:

```
array[ start_index , end_index ]
```

To access the value corresponding to a given key in a map, use the following syntax:

```
map[ key ]
```

The following table describes the arguments in the syntax:

Argument	Description
array	Array. The array from which you want to access one or more elements. You can enter any valid transformation expression that evaluates to an array.
index	Integer. The position of the element that you want to access. For example, an index of 0 indicates the first element in an array.
start_index	Integer. The starting index in a range of elements that you want to access. The subscript operator includes the element that the starting index represents.
end_index	Integer. The ending index in a range of elements that you want to access. The subscript operator excludes the element that the ending index represents.
map	Map. The map from which you want to retrieve the value corresponding to a key.
key	Data type of the key. The key element for which you want to retrieve the value. You can enter any valid transformation expression that evaluates to a key value of the map data.

You can use an expression for the index that returns an integer value. If the expression returns a negative value, the index is considered to be 0.

If the specified index is greater than the size of the array minus 1, the index accesses the final element in the array.

Return Value

If you specify an index, the expression returns the element in the array. The return type is the same as the data type of the element in the specified array.

If you specify two indices separated by a comma, such as `[i,j]`, the expression returns an array of the elements from `i` to `j-1`. If `i` is greater than `j` or the size of the array, the expression returns an empty array. The type configuration of the subarray that the expression returns is the same as the type configuration of the specified array.

If you specify a key, the expression returns the value associated with the key in the map. The return type is the same as the data type of the value in the specified map.

Nulls

If the index in the subscript is greater than the size of the array, the subscript operator returns a NULL value.

If the index is NULL, the subscript operator returns a NULL value. If you specify multiple indices such as `[i,j]` and either `i` or `j` is NULL, the expression returns NULL.

If the array is NULL, the subscript operator returns a NULL value.

If the key does not exist in the map, the subscript operator returns a NULL value.

Examples

You have the following array with string elements:

```
drinks = [‘milk’, ‘coffee’, ‘tea’, ‘chai’]
```

The following expressions use a subscript operator to access string elements from the array:

Input Value	RETURN VALUE
drinks[0]	'milk'
drinks[2]	'tea'
drinks[NULL]	NULL
drinks[1,3]	['coffee','tea']
drinks[2,NULL]	NULL
drinks[3,1]	[ ]

You have the following map with key-value elements of type string-string:

```
country_currency = [‘England’ -> ‘Pound’, ‘France’ -> ‘Euro’, ‘Japan’ -> ‘Yen’>, ‘USA’ -
> ‘Dollar’]
```

Input Value	RETURN VALUE
country_currency [‘Japan’]	‘Yen’
country_currency [‘India’]	NULL
country_currency [‘England’]	‘Pound’

Dot Operator

Use a dot operator to access an element in a struct. You can also use a dot operator in an array of structs to access elements from each struct in the array.

Syntax

To access an element in a struct, use the following syntax:

```
struct.element
```

To access an element in an array of structs, use the following syntax:

```
array_of_structs.element
```

The following table describes the arguments in the syntax:

Argument	Description
struct	Struct. The struct from which you want to access an element. You can enter any valid transformation expression that evaluates to a struct.
array_of_structs	Array with struct elements. The array from which you want to access elements in each struct. You can enter any valid transformation expression that evaluates to an array.
element	The name of the struct element that you want to access.

Return Value

If you use the dot operator on a struct, the expression returns the element in the struct. The return type is the same as the data type of the element in the specified struct.

If you use the dot operator on an array of structs, the expression returns an array that contains the specified element in each struct.

Nulls

If the element in the struct has a NULL value, the expression returns NULL.

If the struct is NULL, the expression returns NULL.

Examples

You have the following struct:

```
location{
    street: NULL
    city : 'NEWYORK'
    state: 'NY'
    zip : 12345
}
```

The following expressions use a dot operator to access elements in the struct:

Input Value	RETURN VALUE
location.street	NULL
location.city	'NEWYORK'
location.state	'NY'
location.zip	12345

You can also use a dot operator to access elements in an array of structs.

For example, you have the following array with three elements of type struct and each struct has three elements:

```
employee_info_array = [

 derrick_struct{
     name: 'Derrick'
     city: NULL
     state: 'NY'
 },

 kevin_struct{
```

```
        name: 'Kevin'
        city: 'Redwood City'
        state: 'CA'
    },

    lauren_struct{
        name: 'Lauren'
        city: 'Woodcliff Lake'
        state: NULL
    }

   ]
```

The following expressions use a dot operator to access the string elements in each struct of the array:

Input Value	RETURN VALUE
employee_info_array.name	['Derrick','Kevin','Lauren']
employee_info_array.city	[NULL,'Redwood City','Woodcliff Lake']
employee_info_array.state	['NY','CA',NULL]

Complex Operators for Nested Data Types

A nested data type contains elements of complex data types. Use a combination of complex operators to access elements in nested data types.

When an array or a struct contains elements of type array or struct, use a combination of complex operators to access the elements. You can access elements in multidimensional arrays, arrays with struct elements, structs with array elements, and structs with struct elements.

Multidimensional Array

A multidimensional array is an array of arrays, which can have up to five levels of nesting. You can use subscript operators to access arrays at any level or specific elements in an array at the innermost level.

You can use subscript operators to return the following values:

- A specific element in an array at the innermost level.
- One or more arrays at any level.
- A subset of one or more arrays at any level.

To access a specific element in an array at the innermost level, you use more than one subscript operator. The number of dimensions in a multidimensional array determines the number of subscript operators to use. Each subscript operator must contain one index value. The data type of the return value is the same as the data type of the elements in the array.

For example, in a two-dimensional array, you use two subscript operators. The first subscript operator determines which one-dimensional array to access. The second subscript operator determines which element to access within the array.

The following two-dimensional array contains three arrays and each array contains elements of type string:

```
menu_array = [
 ['milk','coffee','tea','chai'],
 ['ham','turkey',NULL],
 ['caesar','cobb','greek','chipotle']
]
```

The following expressions use two subscript operators to access a specific element from each one-dimensional array within the `menu_array`:

Input Value	RETURN VALUE
`menu_array[0][1]`	`'coffee'`
`menu_array[2][3]`	`'chipotle'`
`menu_array[1][2]`	`NULL`

The following expressions use a single subscript operator to return one-dimensional arrays in the `menu_array`:

Input Value	RETURN VALUE
`menu_array[0]`	`['milk','coffee','tea','chai']`
`menu_array[0,2]`	`[ ['milk','coffee','tea','chai'], ['ham','turkey',NULL] ]`
`menu_array[1,0]`	`[ ]`
`menu_array[NULL,2]`	`NULL`

The following expressions use two subscript operators to return a subset of arrays within the `menu_array`:

Input Value	RETURN VALUE
`menu_array[0][0,2]`	`['milk','coffee']`
`menu_array[2][0,3]`	`['caesar','cobb','greek']`
`menu_array[0,2][0,3]`	`[ ['milk','coffee','tea'], ['ham','turkey',NULL] ]`

Array with Struct Elements

An array with struct elements is an array of structs. Use a combination of subscript and dot operators to access an element in a struct that is within an array.

To access an element in a struct within an array, use a subscript operator followed by a dot operator. You can also reverse the order of the operators. Return values are the same regardless of the order of the operators. Based on the order of the complex operators, the element is accessed as follows:

You use a subscript operator followed by a dot operator.

The subscript operator first accesses the indexed element in the array and returns a struct. Then, the dot operator accesses an element within the struct.

You use a dot operator followed by a subscript operator.

The dot operator locates elements with the same name from each of the structs and returns an array. Then, the subscript operator accesses an element within the array.

For example, you have the following array, `employee_info_array`:

```
employee_info_array = [
derrick_struct{
    name: 'Derrick'
    city: NULL
    state: 'NY'
},

kevin_struct{
    name: 'Kevin'
    city: 'Redwood City'
    state: 'CA'
},

lauren_struct{
    name: 'Lauren'
    city: 'Woodcliff Lake'
    state: NULL
}
]
```

The following expressions use a subscript operator followed by a dot operator on the array `employee_info_array`:

Input Value	RETURN VALUE
`employee_info_array[0].name`	`'Derrick'`
`employee_info_array[1].city`	`'Redwood City'`
`employee_info_array[2].state`	`NULL`

When you use a dot operator first, the dot operator returns an array with elements of the same name from each struct. For example, the following expressions show the return value when you use a dot operator:

Input Value	RETURN VALUE
`employee_info_array.name`	`['Derrick','Kevin','Lauren']`
`employee_info_array.city`	`[NULL,'Redwood City','Woodcliff Lake']`
`employee_info_array.state`	`['NY','CA',NULL]`

Then, the subscript operator accesses an element in the returned array. The following expressions use a dot operator followed by a subscript operator:

Input Value	RETURN VALUE
`employee_info_array.name[0]`	`'Derrick'`
`employee_info_array.city[1]`	`'Redwood City'`
`employee_info_array.state[2]`	`NULL`

Note that the return values are the same whether you use a subscript operator or a dot operator first. For example, the expressions `employee_info_array[0].name` and `employee_info_array.name[0]` have the same return value `'Derrick'`.

Struct with Array Elements

To access elements in an array that is within a struct, use a dot operator followed by a subscript operator. The dot operator first accesses the specified array element in a struct. Then, the subscript operator accesses elements in the array based on the index value.

For example, you have the following struct with the array elements `drinks, sandwiches,` and `salads`.

```
menu_struct{
drinks: ['milk','coffee','tea','chai']
sandwiches: ['ham','turkey',NULL]
salads: ['caesar','cobb','greek','chipotle']
}
```

When you use the expression `menu_struct.drinks[0]`, the dot operator first accesses the array element `drinks`. Then, the subscript operator accesses the element at position 0 in the array `drinks: ['milk','coffee','tea','chai']`. The return value is `milk`.

The following expressions use a dot operator followed by a subscript operator to access elements from the arrays in the struct `menu_struct`:

Input Value	RETURN VALUE
`menu_struct.drinks[1]`	`'coffee'`
`menu_struct.sandwiches[2]`	`NULL`
`menu_struct.salads[3]`	`'chipotle'`
`menu_struct.drinks[0,3]`	`['milk','coffee','tea']`

Struct with Struct Elements

A struct that contains one or more levels of structs is a nested struct. You can use dot operators to access structs at any level or specific elements in a struct at the innermost level.

You can use dot operators to return the following values:

- A specified element in a struct at the innermost level.
- One or more structs at any level.

To access a specific element in a struct at the innermost level, you use more than one dot operator. The number of levels in a nested struct determines the number of dot operators to use. The data type of the return value is the same as the data type of the element in the struct. For example, in a nested struct of two levels, you use two dot operators. The first dot operator accesses the specified child struct element in a parent struct. Then, the second dot operator accesses elements in the child struct.

The following example uses a struct `employee_info_struct` that contains two child structs `home_address_info` and `department_info`:

```
employee_info_struct{

emp_name: 'Derrick'

home_address_info{
    city: 'New York'
    state: NULL
```

```
department_info{
    NULL
}

}
```

The following expressions use dot operators to access elements from the struct `employee_info_struct`:

Input Value	RETURN VALUE
`employee_info_struct.emp_name`	`'Derrick'`
`employee_info_struct.home_address_info`	`{ city: 'New York' state: NULL }`
`employee_info_struct.department_info`	`NULL`
`employee_info_struct.home_address_info.city`	`'New York'`
`employee_info_struct.home_address_info.state`	`NULL`

Arithmetic Operators

Use arithmetic operators to perform mathematical calculations on numeric data.

The following table lists the arithmetic operators in order of precedence in the transformation language:

Operator	Meaning
+, -	Unary plus and minus. Unary plus indicates a positive value. Unary minus indicates a negative value.
*, /, %	Multiplication, division, modulus. A modulus is the remainder after dividing two integers. For example, 13 % 2 = 1 because 13 divided by 2 equals 6 with a remainder of 1.
+, -	Addition, subtraction. The addition operator (+) does not concatenate strings. To concatenate strings, use the string operator ||. To perform arithmetic on date values, use the date functions.

If you perform arithmetic on a null value, the function returns NULL.

When you use arithmetic operators in an expression, all of the operands in the expression must be numeric. For example, the expression `1 + '1'` is not valid because it adds an integer to a string. The expression `1.23 + 4 / 2` is valid because all of the operands are numeric.

Note: The transformation language provides built-in date functions that let you perform arithmetic on date/time values.

String Operators

Use the || string operator to concatenate two strings. The || operator converts operands of any datatype (except Binary) to String datatypes before concatenation:

Input Value	Return Value
'alpha' || 'betical'	alphabetical
'alpha' || 2	alpha2
'alpha' || NULL	alpha

The || operator includes leading and trailing blanks. Use the LTRIM and RTRIM functions to trim leading and trailing blanks before concatenating two strings.

Nulls

The || operator ignores null values. However, if both values are NULL, the || operator returns NULL.

Example

The following example shows an expression that concatenates employee first names and employee last names from two columns. This expression removes the spaces from the end of the first name and the beginning of the last name, concatenates a space to the end of each first name, then concatenates the last name:

```
LTRIM( RTRIM( EMP_FIRST ) || ' ' || LTRIM( EMP_LAST ))
```

EMP_FIRST	EMP_LAST	RETURN VALUE
' Alfred'	' Rice '	Alfred Rice
' Bernice'	' Kersins'	Bernice Kersins
NULL	' Proud'	Proud
' Curt'	NULL	Curt
NULL	NULL	NULL

Note: You can also use the CONCAT function to concatenate two string values. The || operator, however, produces the same results in less time.

Comparison Operators

Use comparison operators to compare character or numeric strings, manipulate data, and return a TRUE (1) or FALSE (0) value.

The following table lists the comparison operators in the transformation language:

Operator	Meaning
=	Equal to.
>	Greater than.
<	Less than.
>=	Greater than or equal to.
<=	Less than or equal to.
<>	Not equal to.
!=	Not equal to.
^=	Not equal to.

Use the greater than (>) and less than (<) operators to compare numeric values or return a range of rows based on the sort order for a primary key in a particular port.

When you use comparison operators in an expression, the operands must be the same datatype. For example, the expression 123.4 > '123' is not valid because the expression compares a decimal with a string. The expressions 123.4 > 123 and 'a' != 'b' are valid because the operands are the same datatype.

If you compare a value to a null value, the result is NULL.

If a filter condition evaluates to NULL, the Integration Service returns NULL.

Comparing Complex Data Types

You can use the equal to (=) and not equal to (!=) operators to compare complex data types such as arrays or structs.

For two arrays to be equivalent, the following conditions must apply:

- The array elements must be of the same data type.
- The arrays must be the same size.
- The entry at each index must be the same.

For example, you have the following arrays:

```
A = [1, 2, 3]
B = [1, 2, 3]
```

You can make the following comparison:

```
A = B
```

RETURN VALUE: TRUE (1)

Both arrays are the same size and the entry at each index is the same such that `A[0]=B[0], A[1]=B[1], and A[2]=B[2].`

When you compare two structs, the structs are equivalent if they meet the following conditions:

- The corresponding struct elements must be of the same data type.
- The structs must have the same data.

If these conditions are satisfied, the two structs are equivalent even if the struct elements have different names.

For example, you have the following structs:

```
struct1 {
    name:'Paul'
    zip:10004
}

struct2 {
    firstname:'Paul'
    zip1:10004
}
```

You can make the following comparison:

```
struct1 = struct2
```

RETURN VALUE: TRUE (1)

Logical Operators

Use logical operators to manipulate numeric data. Expressions that return a numeric value evaluate to TRUE for values other than 0, FALSE for 0, and NULL for NULL.

The following table lists the logical operators in the transformation language:

Operator	Meaning
NOT	Negates result of an expression. For example, if an expression evaluates to TRUE, the operator NOT returns FALSE. If an expression evaluates to FALSE, NOT returns TRUE.
AND	Joins two conditions and returns TRUE if both conditions evaluate to TRUE. Returns FALSE if one condition is not true.
OR	Connects two conditions and returns TRUE if any condition evaluates to TRUE. Returns FALSE if both conditions are not true.

Nulls

Expressions that combine a null value with a Boolean expression produce results that are ANSI compliant. For example, the Data Integration Service produces the following results:

- NULL AND TRUE = NULL
- NULL AND FALSE = FALSE

CHAPTER 4

Variables

This chapter includes the following topics:

- Built-in Variables, 34
- Local Variables, 34

Built-in Variables

The transformation language provides the built-in variable SYSDATE that returns the system date. You can use SYSDATE in an expression. For example, you can use SYSDATE in a DATE_DIFF function.

SYSDATE

SYSDATE returns the current date and time up to seconds on the node that process data for each row passing through the transformation. SYSDATE is stored as a transformation date/time datatype value.

Example

The following expression uses SYSDATE to find orders that have shipped in the last two days and flag them for insertion. Using DATE_DIFF, the Data Integration Service subtracts DATE_SHIPPED from the system date, returning the difference between the two dates. Because DATE_DIFF returns a double value, the expression truncates the difference. It then compares the result to the integer literal 2. If the result is greater than 2, the expression flags the rows for rejection. If the result is 2 or less, it flags them for insertion.

```
IIF( TRUNC( DATE_DIFF( SYSDATE, DATE_SHIPPED, 'DD' ),
0 ) > 2, DD_REJECT, DD_INSERT
```

Local Variables

If you use local variables in a mapping, use them in any transformation expression in the mapping. For example, if you use a complex tax calculation throughout a mapping, you might want to write the expression once and designate it as a variable. This increases performance since the Data Integration Service performs the calculation only once.

Local variables are useful when used with stored procedure expressions to capture multiple return values.

CHAPTER 5

Dates

This chapter includes the following topics:

- Dates Overview, 35
- Date Format Strings, 39
- TO_CHAR Format Strings, 40
- TO_DATE and IS_DATE Format Strings, 43
- Understanding Date Arithmetic, 47

Dates Overview

The transformation language provides a set of date functions and built-in date variables to perform transformations on dates. With the date functions, you can round, truncate, or compare dates, extract one part of a date, or perform arithmetic on a date. You can pass any value with a date datatype to a date function.

Use date variables to capture the current date on the node hosting the Data Integration Service.

The transformation language also provides the following sets of format strings:

- **Date format strings.** Use with date functions to specify the parts of a date.
- **TO_CHAR format strings.** Use to specify the format of the return string.
- **TO_DATE and IS_DATE format strings.** Use to specify the format of a string you want to convert to a date or test.

Date/Time Datatype

Informatica uses generic datatypes to transform data from different sources. These transformation datatypes include a Date/Time datatype that supports datetime values up to the nanosecond. Informatica stores dates internally in binary format.

Date functions accept datetime values only. To pass a string to a date function, first use TO_DATE to convert it to a datetime value. For example, the following expression converts a string port to datetime values and then adds one month to each date:

```
ADD_TO_DATE( TO_DATE( STRING_PORT, 'MM/DD/RR'), 'MM', 1 )
```

You can use dates between 1 A.D. and 9999 A.D in the Gregorian calendar system.

Julian Day, Modified Julian Day, and the Gregorian Calendar

You can use dates in the Gregorian calendar system only. Dates in the Julian calendar are called Julian *dates* and are not supported in Informatica. This term should not be confused with Julian *Day* or with Modified Julian Day.

You can manipulate Modified Julian Day (MJD) formats using the J format string. The MJD for a given date is the number of days to that date since Jan 1 4713 B.C. 00:00:00 (midnight). By definition, MJD includes a time component expressed as a decimal, which represents some fraction of 24 hours. The J format string does not convert this time component.

For example, the following TO_DATE expression converts strings in the SHIP_DATE_MJD_STRING port to date values in the default date format:

```
TO_DATE (SHIP_DATE_MJD_STR, 'J')
```

SHIP_DATE_MJD_STR	RETURN_VALUE
2451544	Dec 31 1999 00:00:00.000000000
2415021	Jan 1 1900 00:00:00.000000000

SHIP_DATE_MJD_STR	RETURN_VALUE
2451544	Dec 31 1999 00:00:00.000000000
2415021	Jan 1 1900 00:00:00.000000000

Because the J format string does not include the time portion of a date, the return values have the time set to 00:00:00.000000000.

You can also use the J format string in TO_CHAR expressions. For example, use the J format string in a TO_CHAR expression to convert date values to MJD values expressed as strings. For example:

```
TO_CHAR(SHIP_DATE, 'J')
```

SHIP_DATE	RETURN_VALUE
Dec 31 1999 23:59:59	2451544
Jan 1 1900 01:02:03	2415021

Note: The Data Integration Service ignores the time portion of the date in a TO_CHAR expression.

Dates in the Year 2000

All transformation language date functions support the year 2000. Informatica Developer supports dates between 1 A.D. and 9999 A.D.

RR Format String

The transformation language provides the RR format string to convert strings with two-digit years to dates. Using TO_DATE and the RR format string, you can convert a string in the format MM/DD/RR to a date. The RR format string converts data differently depending on the current year.

- **Current Year Between 0 and 49.** If the current year is between 0 and 49 (such as 2003) and the source string year is between 0 and 49, the Data Integration Service returns the current century plus the two-digit year from the source string. If the source string year is between 50 and 99, the Integration Service returns the previous century plus the two-digit year from the source string.
- **Current Year Between 50 and 99.** If the current year is between 50 and 99 (such as 1998) and the source string year is between 0 and 49, the Data Integration Service returns the next century plus the two-digit year from the source string. If the source string year is between 50 and 99, the Data Integration Service returns the current century plus the specified two-digit year.

The following table summarizes how the RR format string converts to dates:

Current year	Source year	RR Format String Returns
0-49	0-49	Current century
0-49	50-99	Previous century
50-99	0-49	Next century
50-99	50-99	Current century

Example

The following expression produces the same return values for any current year between 1950 and 2049:

```
TO_DATE( ORDER_DATE, 'MM/DD/RR' )
```

ORDER_DATE	RETURN_VALUE
'04/12/98'	04/12/1998 00:00:00.000000000
'11/09/01'	11/09/2001 00:00:00.000000000

Difference Between the YY and RR Format Strings

Informatica Developer also provides a YY format string. Both the RR and YY format strings specify two-digit years. The YY and RR format strings produce identical results when used with all date functions except TO_DATE. In TO_DATE expressions, RR and YY produce different results.

The following table shows the different results each format string returns:

String	Current Year	TO_DATE(String, 'MM/DD/RR')	TO_DATE(String, 'MM/DD/YY')
04/12/98	1998	04/12/1998 00:00:00.000000000	04/12/1998 00:00:00.000000000
11/09/01	1998	11/09/2001 00:00:00.000000000	11/09/1901 00:00:00.000000000

String	Current Year	TO_DATE(String, 'MM/DD/RR')	TO_DATE(String, 'MM/DD/YY')
04/12/98	2003	04/12/1998 00:00:00.000000000	04/12/2098 00:00:00.000000000
11/09/01	2003	11/09/2001 00:00:00.000000000	11/09/2001 00:00:00.000000000

For dates in the year 2000 and beyond, the YY format string produces less meaningful results than the RR format string. Use the RR format string for dates in the twenty-first century.

Dates in Relational Databases

In general, dates stored in relational databases contain a date and time value. The date includes the month, day, and year, while the time might include the hours, minutes, seconds, and sub-seconds. You can pass datetime data to any of the date functions.

Dates in Flat Files

Use the TO_DATE function to convert strings to datetime values. You can also use IS_DATE to check if a string is a valid date before converting it with TO_DATE. The transformation language date functions accept date values only. To pass a string to a date function, you must first use the TO_DATE function to convert it to a transformation Date/Time datatype.

Default Date Format

The Data Integration Service uses a default date format to store and manipulate strings that represent dates. To specify the default date format, enter a date format in the DateTime Format String attribute in the data viewer configuration. By default, the date format is MM/DD/YYYY HH24:MI:SS.US.

Because Informatica stores dates in binary format, the Data Integration Service uses the default date format when you perform the following actions:

- **Convert a date to a string by connecting a date/time port to a string port.** The Data Integration Service converts the date to a string in the date format defined in the data viewer configuration.
- **Convert a string to a date by connecting a string port to a date/time port.** The Data Integration Service expects the string values to be in the date format defined by the data viewer configuration. If an input value does not match this format, or if it is an invalid date, the Data Integration Service skips the row. If the string is in this format, the Data Integration Service converts the string to a date value.
- **Use TO_CHAR(date, [format_string]) to convert dates to strings.** If you omit the format string, the Data Integration Service returns the string in the date format defined in the data viewer configuration. If you specify a format string, the Data Integration Service returns a string in the specified format.
- **Use TO_DATE(date, [format_string]) to convert strings to dates.** If you omit the format string, the Data Integration Service expects the string in the date format defined in the data viewer configuration. If you specify a format string, the Data Integration Service expects a string in the specified format.

The default date format of MM/DD/YYYY HH24:MI:SS.US consists of:

- Month (January = 01, September = 09)
- Day (of the month)
- Year (expressed in four digits, such as 1998)
- Hour (in 24-hour format, for example, 12:00:00AM = 0, 1:00:00AM = 1, 12:00:00PM = 12, 11:00:00PM = 23)
- Minutes

- Seconds
- Microseconds

Date Format Strings

You can evaluate input dates using a combination of format strings and date functions. Date format strings are not internationalized and must be entered in predefined formats as listed in the following table.

The following table summarizes the format strings to specify a part of a date:

Format String	Description
D, DD, DDD, DAY, DY, J	Days (01-31). Use any of these format strings to specify the entire day portion of a date. For example, if you pass 12-APR-1997 to a date function, use any of these format strings specify 12.
HH, HH12, HH24	Hour of day (0-23), where 0 is 12 AM (midnight). Use any of these formats to specify the entire hour portion of a date. For example, if you pass the date 12-APR-1997 2:01:32 PM, use HH, HH12, or HH24 to specify the hour portion of the date.
MI	Minutes (0-59).
MM, MON, MONTH	Month (01-12). Use any of these format strings to specify the entire month portion of a date. For example, if you pass 12-APR-1997 to a date function, use MM, MON, or MONTH to specify APR.
MS	Milliseconds (0-999).
NS	Nanoseconds (0-999999999).
SS, SSSS	Seconds (0-59).
US	Microseconds (0-999999).
Y, YY, YYY, YYYY, RR	Year portion of date (0001 to 9999). Use any of these format strings to specify the entire year portion of a date. For example, if you pass 12-APR-1997 to a date function, use Y, YY, YYY, or YYYY to specify 1997.

Note: The format string is not case sensitive. It must always be enclosed within single quotation marks.

The following table describes date functions that use date format strings to evaluate input dates:

Function	Description
ADD_TO_DATE	The part of the date you want to change.
DATE_DIFF	The part of the date to use to calculate the difference between two dates.
GET_DATE_PART	The part of the date you want to return. This function returns an integer value based on the default date format.
IS_DATE	The date you want to check.

Function	Description
ROUND	The part of the date you want to round.
SET_DATE_PART	The part of the date you want to change.
SYSTIMESTAMP	The timestamp precision.
TO_CHAR (Dates)	The character string.
TO_DATE	The character string.
TRUNC (Dates)	The part of the date you want to truncate.

TO_CHAR Format Strings

The TO_CHAR function converts a Date/Time datatype to a string with the format you specify. You can convert the entire date or a part of the date to a string. You might use TO_CHAR to convert dates to strings, changing the format for reporting purposes.

TO_CHAR is generally used when the target is a flat file or a database that does not support a Date/Time datatype.

The following table summarizes the format strings for dates in the function TO_CHAR:

Format String	Description
AM, A.M., PM, P.M.	Meridian indicator. Use any of these format strings to specify AM and PM hours. AM and PM return the same values as A.M. and P.M.
D	Day of week (1-7), where Sunday equals 1.
DAY	Name of day, including up to nine characters (for example, Wednesday).
DD	Day of month (01-31).
DDD	Day of year (001-366, including leap years).
DY	Abbreviated three-character name for a day (for example, Wed).
HH, HH12	Hour of day (01-12).
HH24	Hour of day (00-23), where 00 is 12AM (midnight).
J	Modified Julian Day. Converts the calendar date to a string equivalent to its Modified Julian Day value, calculated from Jan 1, 4713 00:00:00 B.C. It ignores the time component of the date. For example, the expression TO_CHAR(SHIP_DATE, 'J') converts Dec 31 1999 23:59:59 to the string 2451544.
MI	Minutes (00-59).

Format String	Description
MM	Month (01-12).
MONTH	Name of month, including up to nine characters (for example, January).
MON	Abbreviated three-character name for a month (for example, Jan).
MS	Milliseconds (0-999).
NS	Nanoseconds (0-999999999).
Q	Quarter of year (1-4), where January to March equals 1.
RR	Last two digits of a year. The function removes the leading digits. For example, if you use 'RR' and pass the year 1997, TO_CHAR returns 97. When used with TO_CHAR, 'RR' produces the same results as, and is interchangeable with, 'YY.' However, when used with TO_DATE, 'RR' calculates the closest appropriate century and supplies the first two digits of the year.
SS	Seconds (00-59).
SSSSS	Seconds since midnight (00000 - 86399). When you use SSSSS in a TO_CHAR expression, the Data Integration Service only evaluates the time portion of a date. For example, the expression TO_CHAR(SHIP_DATE, 'MM/DD/YYYY SSSSS') converts 12/31/1999 01:02:03 to 12/31/1999 03723.
US	Microseconds (0-999999).
Y	Last digit of a year. The function removes the leading digits. For example, if you use 'Y' and pass the year 1997, TO_CHAR returns 7.
YY	Last two digits of a year. The function removes the leading digits. For example, if you use 'YY' and pass the year 1997, TO_CHAR returns 97.
YYY	Last three digits of a year. The function removes the leading digits. For example, if you use 'YYY' and pass the year 1997, TO_CHAR returns 997.
YYYY	Entire year portion of date. For example, if you use 'YYYY' and pass the year 1997, TO_CHAR returns 1997.
W	Week of month (1-5), where week 1 starts on the first day of the month and ends on the seventh, week 2 starts on the eighth day and ends on the fourteenth day. For example, Feb 1 designates the first week of February.
WW	Week of year (01-53), where week 01 starts on Jan 1 and ends on Jan 7, week 2 starts on Jan 8 and ends on Jan 14, and so on.
- / . ; :	Punctuation that displays in the output. You might use these symbols to separate date parts. For example, you create the following expression to separate date parts with a period: TO_CHAR(DATES, 'MM.DD.YYYY').

Format String	Description
“text”	Text that displays in the output. For example, if you create an output port with the expression: TO_CHAR(DATES, ‘MM/DD/YYYY “Sales Were Up”’) and pass the date Apr 1 1997, the function returns the string ‘04/01/1997 Sales Were Up’. You can enter multibyte characters that are valid in the repository code page.
“”	Use double quotation marks to separate ambiguous format strings, for example D“”DDD. The empty quotation marks do not appear in the output.

Note: The format string is not case sensitive. It must always be enclosed within single quotation marks.

Examples

The following examples show the J, SSSSS, RR, and YY format strings. See the individual functions for more examples.

Note: The Data Integration Service ignores the time portion of the date in a TO_CHAR expression.

J Format String

Use the J format string in a TO_CHAR expression to convert date values to MJD values expressed as strings. For example:

```
TO_CHAR(SHIP_DATE, 'J')
```

SHIP_DATE	RETURN_VALUE
Dec 31 1999 23:59:59	2451544
Jan 1 1900 01:02:03	2415021

SSSSS Format String

You can also use the format string SSSSS in a TO_CHAR expression. For example, the following expression converts the dates in the SHIP_DATE port to strings representing the total seconds since midnight:

```
TO_CHAR( SHIP_DATE, 'SSSSS')
```

SHIP_DATE	RETURN_VALUE
12/31/1999 01:02:03	3723
09/15/1996 23:59:59	86399

RR Format String

The following expression converts dates to strings in the format MM/DD/YY:

```
TO_CHAR( SHIP_DATE, 'MM/DD/RR')
```

SHIP_DATE	RETURN_VALUE
12/31/1999 01:02:03	12/31/99
09/15/1996 23:59:59	09/15/96
05/17/2003 12:13:14	05/17/03

YY Format String

In TO_CHAR expressions, the YY format string produces the same results as the RR format string. The following expression converts dates to strings in the format MM/DD/YY:

```
TO_CHAR( SHIP_DATE, 'MM/DD/YY')
```

SHIP_DATE	RETURN_VALUE
12/31/1999 01:02:03	12/31/99
09/15/1996 23:59:59	09/15/96
05/17/2003 12:13:14	05/17/03

TO_DATE and IS_DATE Format Strings

The TO_DATE function converts a string with the format you specify to a datetime value. TO_DATE is generally used to convert strings from flat files to datetime values. TO_DATE format strings are not internationalized and must be entered in the predefined formats.

Note: TO_DATE and IS_DATE use the same set of format strings.

When you create a TO_DATE expression, use a format string for each part of the date in the source string. The source string format and the format string must match. The date separator need not match for date validation to take place. If any part does not match, the Data Integration Service does not convert the string, and it skips the row. If you omit the format string, the source string must be in the date format specified in the data viewer configuration.

IS_DATE indicates whether a value is a valid date. A valid date is any string in the date format specified in the data viewer configuration. If the strings that you want to test are not in the specified date format, use the format of the strings listed in "TO_DATE and IS_DATE Format Strings" table. If the format of a string does not match the specified format or if the string does not represent a valid date, the function returns FALSE (0). If the format of the string matches the specified format of the string and is a valid date, the function returns TRUE (1). IS_DATE format strings are not internationalized and must be entered in one of the formats listed in the following table.

The following table lists the format strings for the functions TO_DATE and IS_DATE:

Table 1. TO_DATE and IS_DATE Format Strings

Format String	Description
AM, a.m., PM, p.m.	Meridian indicator. Use any of these format strings to specify AM and PM hours. AM and PM return the same values as do a.m. and p.m.
DAY	Name of day, including up to nine characters (for example, Wednesday). The DAY format string is not case sensitive.
DD	Day of month (1-31).
DDD	Day of year (001-366, including leap years).
DY	Abbreviated three-character name for a day (for example, Wed). The DY format string is not case sensitive.
HH, HH12	Hour of day (1-12).
HH24	Hour of day (0-23), where 0 is 12AM (midnight).
J	Modified Julian Day. Convert strings in MJD format to date values. It ignores the time component of the source string, assigning all dates the time of 00:00:00.000000000. For example, the expression TO_DATE('2451544', 'J') converts 2451544 to Dec 31 1999 00:00:00.000000000.
MI	Minutes (0-59).
MM	Month (1-12).
MONTH	Name of month, including up to nine characters (for example, August). Case does not matter.
MON	Abbreviated three-character name for a month (for example, Aug). Case does not matter.
MS	Milliseconds (0-999).
NS	Nanoseconds (0-999999999).
RR	Four-digit year (for example, 1998, 2034). Use when source strings include two-digit years. Use with TO_DATE to convert two-digit years to four-digit years. - Current Year Between 50 and 99. If the current year is between 50 and 99 (such as 1998) and the year value of the source string is between 0 and 49, the Data Integration Service returns the next century plus the two-digit year from the source string. If the year value of the source string is between 50 and 99, the Data Integration Service returns the current century plus the specified two-digit year. - Current Year Between 0 and 49. If the current year is between 0 and 49 (such as 2003) and the source string year is between 0 and 49, the Data Integration Service returns the current century plus the two-digit year from the source string. If the source string year is between 50 and 99, the Data Integration Service returns the previous century plus the two-digit year from the source string.
SS	Seconds (0-59).

Format String	Description
SSSSS	Seconds since midnight. When you use SSSSS in a TO_DATE expression, the Data Integration Service only evaluates the time portion of a date. For example, the expression TO_DATE(DATE_STR, 'MM/DD/YYYY SSSSS') converts 12/31/1999 3783 to 12/31/1999 01:02:03.
US	Microseconds (0-999999).
Y	The current year on the node running the Data Integration Service with the last digit of the year replaced with the string value.
YY	The current year on the node running the Data Integration Service with the last two digits of the year replaced with the string value.
YYY	The current year on the node running the Data Integration Service with the last three digits of the year replaced with the string value.
YYYY	Four digits of a year. Do not use this format string if you are passing two-digit years. Use the RR or YY format string instead.

Rules and Guidelines for Date Format Strings

Use the following rules and guidelines when you work with date format strings:

- The format of the TO_DATE string must match the format string. If it does not, the Data Integration Service might return inaccurate values or skip the row. For example, if you pass the string '20200512', representing May 12, 2020, to TO_DATE, you must include the format string YYYYMMDD. If you do not include a format string, the Data Integration Service expects the string in the date format specified in the data viewer configuration. Likewise, if you pass a string that does not match the format string, the Data Integration Service returns an error and skips the row. For example, if you pass the string 2020120 to TO_DATE and include the format string YYYYMMDD, the Data Integration Service returns an error and skips the row because the string does not match the format string.
- The format string must be enclosed within single quotation marks.
- The Data Integration Service uses the default date time format specified in the session. Default is MM/DD/YYYY HH24:MI:SS.US. The format string is not case sensitive.

Example

The following examples illustrate the J, RR, and SSSSS format strings. See the individual functions for more examples.

J Format String

The following expression converts strings in the SHIP_DATE_MJD_STRING port to date values in the default date format:

```
TO_DATE (SHIP_DATE_MJD_STR, 'J')
```

SHIP_DATE_MJD_STR	RETURN_VALUE
2451544	Dec 31 1999 00:00:00.000000000
2415021	Jan 1 1900 00:00:00.000000000

Because the J format string does not include the time portion of a date, the return values have the time set to 00:00:00.000000000.

RR Format String

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/RR')
```

DATE_STR	RETURN VALUE
04/01/98	04/01/1998 00:00:00.000000000
08/17/05	08/17/2005 00:00:00.000000000

YY Format String

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/YY')
```

DATE_STR	RETURN VALUE
04/01/98	04/01/1998 00:00:00.000000000
08/17/05	08/17/1905 00:00:00.000000000

Note: For the second row, RR returns the year 2005, but YY returns the year 1905.

SSSSS Format String

The following expression converts strings that include the seconds since midnight to date values:

```
TO_DATE( DATE_STR, 'MM/DD/YYYY SSSSS')
```

DATE_STR	RETURN_VALUE
12/31/1999 3783	12/31/1999 01:02:03.000000000
09/15/1996 86399	09/15/1996 23:59:59.000000000

Understanding Date Arithmetic

The transformation language provides built-in date functions so you can perform arithmetic on datetime values as follows:

- **ADD_TO_DATE.** Add or subtract a specific portion of a date.
- **DATE_DIFF.** Subtract two dates.
- **SET_DATE_PART.** Change one part of a date.

You cannot use numeric arithmetic operators (such as + or -) to add or subtract dates.

The transformation language recognizes leap years and accepts dates between Jan. 1, 0001 00:00:00.000000000 A.D. and Dec. 31, 9999 23:59:59.999999999 A.D.

Note: The transformation language uses the transformation Date/Time datatype to specify date values. You can only use the date functions on datetime values.

CHAPTER 6

Functions

This chapter includes information about function support in the transformation language.

Function Categories

The transformation language provides the following types of functions:

- Aggregate
- Character
- Complex
- Conversion
- Data Cleansing
- Date
- Encoding
- Financial
- Numerical
- Scientific
- Special
- String
- Test
- Window

Aggregate Functions

Aggregate functions return summary values for non-null values in selected ports. With aggregate functions you can:

- Calculate a single value for all rows in a group.
- Return a single value for each group in an Aggregator transformation.
- Apply filters to calculate values for specific rows in the selected ports.
- Use operators to perform arithmetic within the function.
- Calculate two or more aggregate values derived from the same source columns in a single pass.

The transformation language includes the following aggregate functions:

- ANY
- AVG
- COLLECT_LIST
- COLLECT_MAP
- COUNT
- FIRST
- LAST
- MAX (Date)
- MAX (Number)
- MAX (String)
- MEDIAN
- MIN (Date)
- MIN (Number)
- MIN (String)
- PERCENTILE
- STDDEV
- SUM
- VARIANCE

If you configure the Data Integration Service to run in Unicode mode, MIN and MAX return values according to the sort order of the code page you specify in the mapping configuration.

You can use aggregate functions in Aggregator transformations. You can nest only one aggregate function within another aggregate function. The Data Integration Service evaluates the innermost aggregate function expression and uses the result to evaluate the outer aggregate function expression. You can set up an Aggregator transformation that groups by ID and nests two aggregate functions as follows:

```
SUM( AVG( earnings ) )
```

where the dataset contains the following values:

ID	EARNINGS
1	32
1	45
1	100
2	65
2	75
2	76
3	21

ID	EARNINGS
3	45
3	99

The return value is 186. The Data Integration Service groups by ID, evaluates the AVG expression, and returns three values. Then it adds the values with the SUM function to get the result.

You can also use aggregate functions as window functions in an Expression transformation. To use an aggregate function as a window function when you run a mapping on the Spark engine, you must configure the transformation for windowing. If you use an aggregate function as a window function, the Expression transformation becomes active.

Aggregate Functions and Nulls

When you configure the Data Integration Service, you can choose how you want to handle null values in aggregate functions. You can have the Data Integration Service treat null values in aggregate functions as NULL or 0.

By default, the Data Integration Service treats null values as NULL in aggregate functions. If you pass an entire port or group of null values to the COUNT function, the function returns 0. If you pass an entire port or group of null values to any other aggregate function, the function returns NULL. You can optionally configure the Data Integration Service if you pass an entire port of null values to an aggregate function to return 0.

Filter Conditions

Use a filter condition to limit the rows returned in a search.

A filter limits the rows returned in a search. You can apply a filter condition to all aggregate functions and to CUME, MOVINGAVG, and MOVINGSUM. The filter condition must evaluate to TRUE, FALSE, or NULL. If the filter condition evaluates to NULL or FALSE, the Data Integration Service does not select the row.

You can enter any valid transformation expression. For example, the following expression calculates the median salary for all employees who make more than $50,000:

```
MEDIAN( SALARY, SALARY > 50000 )
```

You can also use other numeric values as the filter condition. For example, you can enter the following as the complete syntax for the MEDIAN function, including a numeric port:

```
MEDIAN( PRICE, QUANTITY > 0 )
```

In all cases, the Data Integration Service rounds a decimal value to an integer (for example, 1.5 to 2, 1.2 to 1, 0.35 to 0) for the filter condition. If the value rounds to 0, the filter condition returns FALSE. If you do not want to round up a value, use the TRUNC function to truncate the value to an integer:

```
MEDIAN( PRICE, TRUNC( QUANTITY ) > 0 )
```

If you omit the filter condition, the function selects all rows in the port.

Character Functions

The transformation language includes the following character functions:

- ASCII
- CHR

- CHRCODE
- CONCAT
- INITCAP
- INSTR
- LENGTH
- LOWER
- LPAD
- LTRIM
- METAPHONE
- REPLACECHR
- REPLACESTR
- RPAD
- RTRIM
- SOUNDEX
- SUBSTR
- UPPER

The character functions MAX, MIN, LOWER, UPPER, and INITCAP use the code page of the Data Integration Service to evaluate character data.

Complex Functions

A complex function is a type of pre-defined function in which the value of the input or the return type is of a complex data type, such as an array, map, or struct. You can use complex functions in mappings that run on the Spark engine.

The transformation language includes the following complex functions:

- ARRAY
- CAST
- COLLECT_LIST
- COLLECT_MAP
- CONCAT_ARRAY
- EXTRACT_STRUCT
- MAP
- MAP_FROM_ARRAYS
- MAP_KEYS
- MAP_VALUES
- PARSE_JSON
- PARSE_XML
- RESPEC
- SIZE
- STRUCT
- STRUCT_AS

Conversion Functions

The transformation language includes the following conversion functions:

- TO_BIGINT
- TO_CHAR(Number)
- TO_DATE
- TO_DECIMAL
- TO_FLOAT
- TO_INTEGER

Data Cleansing Functions

The transformation language includes a group of functions to eliminate data errors. You can complete the following tasks with data cleansing functions:

- Test input values.
- Convert the datatype of an input value.
- Trim string values.
- Replace characters in a string.
- Encode strings.
- Match patterns in regular expressions.

The transformation language includes the following data cleansing functions:

- GREATEST
- IN
- INSTR
- IS_DATE
- IS_NUMBER
- IS_SPACES
- ISNULL
- LEAST
- LTRIM
- METAPHONE
- REG_EXTRACT
- REG_MATCH
- REG_REPLACE
- REPLACECHR
- REPLACESTR
- RTRIM
- SQL_LIKE
- SOUNDEX
- SUBSTR
- TO_BIGINT

- TO_CHAR
- TO_DATE
- TO_DECIMAL
- TO_FLOAT
- TO_INTEGER

Date Functions

The transformation language includes a group of date functions to round, truncate, or compare dates, extract one part of a date, or perform arithmetic on a date.

You can pass any value with a date datatype to any of the date functions. However, if you want to pass a string to a date function, you must first use the TO_DATE function to convert it to a transformation Date/Time datatype.

The transformation language includes the following date functions:

- ADD_TO_DATE
- DATE_COMPARE
- DATE_DIFF
- GET_DATE_PART
- IS_DATE
- LAST_DAY
- MAKE_DATE_TIME
- MAX
- MIN
- ROUND(Date)
- SET_DATE_PART
- SYSTIMESTAMP
- TO_CHAR(Date)
- TIME_RANGE
- TRUNC(Date)

Several of the date functions include a *format* argument. You must specify one of the transformation language format strings for this argument. Date format strings are not internationalized.

The Date/Time transformation datatype supports dates with precision to the nanosecond.

Encoding Functions

The transformation language includes the following functions for data encryption, compression, encoding, and checksum:

- AES_DECRYPT
- AES_ENCRYPT
- COMPRESS
- CRC32
- DEC_BASE64

- DECOMPRESS
- ENC_BASE64
- MD5

Financial Functions

The transformation language includes the following financial functions:

- FV
- NPER
- PMT
- PV
- RATE

Numeric Functions

The transformation language includes the following numeric functions:

- ABS
- CEIL
- CONV
- CUME
- EXP
- FLOOR
- LN
- LOG
- MAX
- MIN
- MOD
- MOVINGAVG
- MOVINGSUM
- POWER
- RAND
- ROUND
- SIGN
- SQRT
- TRUNC

Scientific Functions

The transformation language includes the following scientific functions:

- COS
- COSH

- SIN
- SINH
- TAN
- TANH

Special Functions

The transformation language includes the following special functions:

- ABORT
- DECODE
- ERROR
- IIF
- LOOKUP
- UUID4
- UUID_UNPARSE

Generally, you use special functions in Expression, Filter, and Update Strategy transformations. You can nest other functions within special functions. You can also nest a special function in an aggregate function.

String Functions

The transformation language includes the following string functions:

- CHOOSE
- INDEXOF
- MAX
- MIN
- REVERSE

Test Functions

The transformation language includes the following test functions:

- ISNULL
- IS_DATE
- IS_NUMBER
- IS_SPACES

Window Functions

The transformation language includes a group of window functions that perform calculations on a set of rows that are related to the current row. The functions calculate a single return value for every input row. You can use window functions in mappings that run on the Spark engine.

The transformation language includes the following window functions:

- LAG
- LEAD

You can use window functions in Expression transformations. If you use a window function in an Expression transformation, the transformation is active.

ABORT

Stops the mapping run, and issues a specified error message to the log. When the Data Integration Service encounters an ABORT function, it stops transforming data at that row. It processes any rows read before the mapping run aborts. The Data Integration Service writes to the target up to the aborted row and then rolls back all uncommitted data to the last commit point.

Use ABORT to validate data. Generally, you use ABORT within an IIF or DECODE function to set rules for aborting a session.

Use the ABORT function for both input and output port default values. You might use ABORT for input ports to keep null values from passing into a transformation. You can also use ABORT to handle any kind of transformation error, including ERROR function calls within an expression. The default value overrides the ERROR function in an expression. If you want to ensure the session stops when an error occurs, assign ABORT as the default value.

If you use ABORT in an expression for an unconnected port, the Data Integration Service does not run the ABORT function.

Syntax

```
ABORT( string )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
string	Required	String. The message you want to display in the log when the mapping run stops. The string can be any length. You can enter any valid transformation expression.

Return Value

NULL.

ABS

Returns the absolute value of a numeric value.

Syntax

```
ABS( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values for which you want to return the absolute values. You can enter any valid transformation expression.

Return Value

Positive numeric value. ABS returns the same datatype as the numeric value passed as an argument. If you pass a Double, it returns a Double. Likewise, if you pass an Integer, it returns an Integer.

NULL if you pass a null value to the function.

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Example

The following expression returns the difference between two numbers as a positive value, regardless of which number is larger:

```
ABS( PRICE - COST )
```

PRICE	COST	RETURN VALUE
250	150	100
52	48	4
169.95	69.95	100
59.95	NULL	NULL
70	30	40
430	330	100
100	200	100

ADD_TO_DATE

Adds a specified amount to one part of a datetime value, and returns a date in the same format as the date you pass to the function. ADD_TO_DATE accepts positive and negative integer values. Use ADD_TO_DATE to change the following parts of a date:

- **Year.** Enter a positive or negative integer in the *amount* argument. Use any of the year format strings: Y, YY, YYY, or YYYY. The following expression adds 10 years to all dates in the SHIP_DATE port:

  ```
  ADD_TO_DATE ( SHIP_DATE, 'YY', 10 )
  ```

- **Month.** Enter a positive or negative integer in the *amount* argument. Use any of the month format strings: MM, MON, MONTH. The following expression subtracts 10 months from each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'MONTH', -10 )
  ```

- **Day.** Enter a positive or negative integer in the *amount* argument. Use any of the day format strings: D, DD, DDD, DY, and DAY. The following expression adds 10 days to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'DD', 10 )
  ```

- **Hour.** Enter a positive or negative integer in the *amount* argument. Use any of the hour format strings: HH, HH12, HH24. The following expression adds 14 hours to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'HH', 14 )
  ```

- **Minute.** Enter a positive or negative integer in the *amount* argument. Use the MI format string to set the minute. The following expression adds 25 minutes to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'MI', 25 )
  ```

- **Seconds.** Enter a positive or negative integer in the *amount* argument. Use the SS format string to set the second. The following expression adds 59 seconds to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'SS', 59 )
  ```

- **Milliseconds.** Enter a positive or negative integer in the *amount* argument. Use the MS format string to set the milliseconds. The following expression adds 125 milliseconds to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'MS', 125 )
  ```

- **Microseconds.** Enter a positive or negative integer in the *amount* argument. Use the US format string to set the microseconds. The following expression adds 2,000 microseconds to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'US', 2000 )
  ```

- **Nanoseconds.** Enter a positive or negative integer in the *amount* argument. Use the NS format string to set the nanoseconds. The following expression adds 3,000,000 nanoseconds to each date in the SHIP_DATE port:

  ```
  ADD_TO_DATE( SHIP_DATE, 'NS', 3000000 )
  ```

Syntax

```
ADD_TO_DATE( date, format, amount )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. Passes the values you want to change. You can enter any valid transformation expression.
format	Required	A format string specifying the portion of the date value you want to change. Enclose the format string within single quotation marks, for example, 'mm'. The format string is not case sensitive.
amount	Required	An integer value specifying the amount of years, months, days, hours, and so on by which you want to change the date value. You can enter any valid transformation expression that evaluates to an integer.

Return Value

Date in the same format as the date you pass to this function.

NULL if a null value is passed as an argument to the function.

Examples

The following expressions all add one month to each date in the DATE_SHIPPED port. If you pass a value that creates a day that does not exist in a particular month, the Data Integration Service returns the last day of the month. For example, if you add one month to Jan 31 1998, the Data Integration Service returns Feb 28 1998.

Also note, ADD_TO_DATE recognizes leap years and adds one month to Jan 29 2000:

```
ADD_TO_DATE( DATE_SHIPPED, 'MM', 1 )
ADD_TO_DATE( DATE_SHIPPED, 'MON', 1 )
ADD_TO_DATE( DATE_SHIPPED, 'MONTH', 1 )
```

DATE_SHIPPED	RETURN VALUE
Jan 12 1998 12:00:30AM	Feb 12 1998 12:00:30AM
Jan 31 1998 6:24:45PM	Feb 28 1998 6:24:45PM
Jan 29 2000 5:32:12AM	Feb 29 2000 5:32:12AM *(Leap Year)*
Oct 9 1998 2:30:12PM	Nov 9 1998 2:30:12PM
NULL	NULL

The following expressions subtract 10 days from each date in the DATE_SHIPPED port:

```
ADD_TO_DATE( DATE_SHIPPED, 'D', -10 )
ADD_TO_DATE( DATE_SHIPPED, 'DD', -10 )
ADD_TO_DATE( DATE_SHIPPED, 'DDD', -10 )
ADD_TO_DATE( DATE_SHIPPED, 'DY', -10 )
ADD_TO_DATE( DATE_SHIPPED, 'DAY', -10 )
```

DATE_SHIPPED	RETURN VALUE
Jan 1 1997 12:00:30AM	Dec 22 1996 12:00AM
Jan 31 1997 6:24:45PM	Jan 21 1997 6:24:45PM
Mar 9 1996 5:32:12AM	Feb 29 1996 5:32:12AM *(Leap Year)*
Oct 9 1997 2:30:12PM	Sep 30 1997 2:30:12PM
Mar 3 1996 5:12:20AM	Feb 22 1996 5:12:20AM
NULL	NULL

The following expressions subtract 15 hours from each date in the DATE_SHIPPED port:

```
ADD_TO_DATE( DATE_SHIPPED, 'HH', -15 )
ADD_TO_DATE( DATE_SHIPPED, 'HH12', -15 )
ADD_TO_DATE( DATE_SHIPPED, 'HH24', -15 )
```

DATE_SHIPPED	RETURN VALUE
Jan 1 1997 12:00:30AM	Dec 31 1996 9:00:30AM
Jan 31 1997 6:24:45PM	Jan 31 1997 3:24:45AM
Oct 9 1997 2:30:12PM	Oct 8 1997 11:30:12PM

DATE_SHIPPED	RETURN VALUE
Mar 3 1996 5:12:20AM	Mar 2 1996 2:12:20PM
Mar 1 1996 5:32:12AM	Feb 29 1996 2:32:12PM *(Leap Year)*
NULL	NULL

Working with Dates

Use the following tips when working with ADD_TO_DATE:

- You can add or subtract any part of the date by specifying a format string and making the *amount* argument a positive or negative integer.
- If you pass a value that creates a day that does not exist in a particular month, the Data Integration Service returns the last day of the month. For example, if you add one month to Jan 31 1998, the Data Integration Service returns Feb 28 1998.
- You can nest TRUNC and ROUND to manipulate dates.
- You can nest TO_DATE to convert strings to dates.
- ADD_TO_DATE changes only one portion of the date, which you specify. If you modify a date so that it changes from standard to daylight savings time, you need to change the hour portion of the date.

AES_DECRYPT

Returns decrypted data to string format. The Data Integration Service uses Advanced Encryption Standard (AES) algorithm with 128-bit and 256-bit encoding. The AES algorithm is a FIPS-approved cryptographic algorithm.

Syntax

```
AES_DECRYPT (value, key, keySize)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Binary datatype. Value you want to decrypt.
key	Required	String datatype. Precision of 16 characters or fewer. You can use mapping variables for the key. Use the same key to decrypt a value that you used to encrypt it.
keySize	Required	Integer datatype. Precision of 16 characters or fewer. You can specify 128, 192, or 256-bit encryption. Default is 128-bit.

Return Value

Decrypted binary value.

NULL if the input value is a null value.

Example

The following example returns decrypted social security numbers. In this example, the Data Integration Service derives the key from the first three numbers of the social security number using the SUBSRT function:

```
AES_DECRYPT (SSN_ENCRYPT, SUBSTR(SSN,1,3), 256)
```

SSN_ENCRYPT	DECRYPTED VALUE
07FB945926849D2B1641E708C85E4390	832-17-1672
9153ACAB89D65A4B81AD2ABF151B099D	832-92-4731
AF6B5E4E39F974B3F3FB0F22320CC60B	832-46-7552
992D6A5D91E7F59D03B940A4B1CBBCBE	832-53-6194
992D6A5D91E7F59D03B940A4B1CBBCBE	832-81-9528

AES_ENCRYPT

Returns data in encrypted format. The Data Integration Service uses Advanced Encryption Standard (AES) algorithm with 128-bit and 256-bit encoding. The AES algorithm is a FIPS-approved cryptographic algorithm.

Use this function to prevent sensitive data from being visible to everyone. For example, to store social security numbers in a data warehouse, use the AES_ENCRYPT function to encrypt the social security numbers to maintain confidentiality.

Syntax

```
AES_ENCRYPT (value, key, keySize)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	String datatype. Value you want to encrypt.
key	Required	String datatype. Precision of 16 characters or fewer. You can use mapping variables for the key.
keySize	Required	Integer datatype. Precision of 16 characters or fewer. You can specify 128, 192, or 256-bit encryption. Default is 128-bit.

Return Value

Encrypted binary value.

NULL if the input is a null value.

Example

The following example returns encrypted values for social security numbers. In this example, the Data Integration Service derives the key from the first three numbers of the social security number using the SUBSTR function:

```
AES_ENCRYPT (SSN, SUBSTR(SSN,1,3), 256)
```

SSN	ENCRYPTED VALUE
832-17-1672	07FB945926849D2B1641E708C85E4390
832-92-4731	9153ACAB89D65A4B81AD2ABF151B099D
832-46-7552	AF6B5E4E39F974B3F3FB0F22320CC60B
832-53-6194	992D6A5D91E7F59D03B940A4B1CBBCBE
832-81-9528	20812B3331676B15A9378000EB900EE3

Tip

If the target does not support binary data, use AES_ENCRYPT with the ENC_BASE64 function to store the data in a format compatible with the database.

ANY

Returns any value found within a selected port or group. Optionally, you can apply a filter to limit the rows the Data Integration Service reads. You can nest only one other aggregate function within ANY.

Syntax

```
ANY( value [, filter_condition ] )
```

The following table describes the arguments for this function:

Argument	Required/ Optional	Description
value	Required	Any data type except Binary. Passes the values for which you want to return any row. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Any value in the port or group. Return value might be different each time.

NULL if all values passed to the function are NULL, all values are empty, or no rows are selected. For example, the filter condition evaluates to FALSE or NULL for all rows.

Example

The following expression returns any row in the ITEM_NAME port with a price greater than $10.00:

```
ANY( ITEM_NAME, ITEM_PRICE > 10 )
```

ITEM_NAME	ITEM_PRICE
Flashlight	35.00
Navigation Compass	8.05
Regulator System	150.00
Flashlight	29.00
Depth/Pressure Gauge	88.00
Vest	31.00

RETURN VALUE: Flashlight

ANY and Complex Data Types

You can use ANY to return a row in a complex port of type array or struct.

For example, you have the following array:

```
emp_phones =
[205-128-6478, 722-515-2889]
[107-081-0961, 718-051-8116]
[344-894-6463, 861-411-8361]
[107-031-0961, NULL]
```

You can use the following expression to return any row in the array port:

```
ANY( emp_phones )
```

RETURN VALUE: [205-128-6478, 722-515-2889]

ARRAY

Generates an array with elements based on the specified arguments.

Syntax

```
ARRAY (array_element1 as any [, array_element2] ...)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
array_element1	Required	Any data type. The element that you want to add to the array. You can enter any valid transformation expression.
array_element2	Optional	Same data type as the array_element1.

If you use the ARRAY function in an output expression for an array port, the data type of the function arguments must match the data type of the array elements specified in the type configuration for the array port.

Return Value

Array.

The data type of the arguments determines the data type of the array elements. For example, if you pass string arguments, the function generates an array of string elements.

Examples

The following expression generates an array of string elements.

```
ARRAY (work_phone, home_phone)
```

work_phone	home_phone	RETURN VALUE
205-128-6478	722-515-2889	[205-128-6478,722-515-2889]
107-081-0961	718-051-8116	[107-081-0961,718-051-8116]
344-894-6463	861-411-8361	[344-894-6463,861-411-8361]
107-031-0961	NULL	[107-031-0961,NULL]

ASCII

When the Data Integration Service uses ASCII mode, the ASCII function returns the numeric ASCII value of the first character of the string passed to the function.

When the Data Integration Service uses Unicode mode, the ASCII function returns the numeric Unicode value of the first character of the string passed to the function. Unicode values fall in the range 0 to 65,535.

You can pass a string of any size to ASCII, but it evaluates only the first character in the string. Before you pass any string value to ASCII, you can parse out the specific character you want to convert to an ASCII or Unicode value. For example, you might use RTRIM or another string-manipulation function. If you pass a numeric value, ASCII converts it to a character string and returns the ASCII or Unicode value of the first character in the string.

This function is identical in behavior to the CHRCODE function. If you use ASCII in existing expressions, they will still work correctly. However, when you create new expressions, use the CHRCODE function instead of the ASCII function.

Syntax

```
ASCII ( string )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
string	Required	Character string. Passes the value you want to return as an ASCII value. You can enter any valid transformation expression.

Return Value

Integer. The ASCII or Unicode value of the first character in the string.

NULL if a value passed to the function is NULL.

Example

The following expression returns the ASCII or Unicode value for the first character of each value in the ITEMS port:

```
ASCII( ITEMS )
```

ITEMS	RETURN VALUE
Flashlight	70
Compass	67
Safety Knife	83
Depth/Pressure Gauge	68
Regulator System	82

AVG

Returns the average of all values in a group of rows. Optionally, you can apply a filter to limit the rows you read to calculate the average. You can nest only one other aggregate function within AVG, and the nested function must return a Numeric datatype.

Syntax

```
AVG( numeric_value [, filter_condition ] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values for which you want to calculate an average. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL or no rows are selected. For example, the filter condition evaluates to FALSE or NULL for all rows.

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Nulls

If a value is NULL, AVG ignores the row. However, if all values passed from the port are NULL, AVG returns NULL.

Group By

AVG groups values based on group by ports you define in the transformation, returning one result for each group.

If there is not a group by port, AVG treats all rows as one group, returning one value.

Example

The following expression returns the average wholesale cost of flashlights:

```
AVG( WHOLESALE_COST, ITEM_NAME='Flashlight' )
```

ITEM_NAME	WHOLESALE_COST
Flashlight	35.00
Navigation Compass	8.05
Regulator System	150.00
Flashlight	29.00
Depth/Pressure Gauge	88.00
Flashlight	31.00

RETURN VALUE: 31.66

Tip

You can perform arithmetic on the values passed to AVG before the function calculates the average. For example:

```
AVG( QTY * PRICE - DISCOUNT )
```

CAST

Renames the elements and changes the data type of each element for the given struct value based on the data type in the specified complex data type definition.

Syntax

```
CAST (:Type.type_definition_library.type_definition, struct_value)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
:Type.type_definition_library.type_definition	Required	The complex data type definition that represents the schema of the struct data. Use the reference qualifier `:Type` to reference the type definition library that contains the complex data type definition.
struct_value	Required	The struct value for which you want to change the data type of the struct elements. You can enter any valid transformation expression that evaluates to a struct.

The data type of the struct value and the data type in the complex data type definition must be compatible.

Return Value

Struct.

Examples

The following expression changes the data types of the elements in the struct port h2_sales based on the data types in the complex data type definition h1_sales_def.

```
CAST (:Type.type_definition_library.h1_sales_def, h2_sales)
```

h1_sales_def	h2_sales	RETURN VALUE

{
q1_total : bigint
q2_total : double
}
	{
	q3_total : int
	q4_total : int
	}
		{
		q1_total : bigint
		q2_total : double
		}

CEIL

Returns the smallest integer greater than or equal to the numeric value passed to this function. For example, if you pass 3.14 to CEIL, the function returns 4. If you pass 3.98 to CEIL, the function returns 4. Likewise, if you pass -3.17 to CEIL, the function returns -3.

Syntax

```
CEIL( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric data type. You can enter any valid transformation expression.

Return Value

Numeric value.

Double value if you pass a numeric value with declared precision greater than 38.

NULL if a value passed to the function is NULL.

Example

The following expression returns the price rounded to the next integer:

```
CEIL( PRICE )
```

PRICE	RETURN VALUE
39.79	40
125.12	126
74.24	75
NULL	NULL
-100.99	-100

Tip: You can perform arithmetic on the values passed to CEIL before CEIL returns the next integer value. For example, if you want to multiply a numeric value by 10 before you calculated the smallest integer greater than the modified value, you might write the function as follows:

```
CEIL( PRICE * 10 )
```

CHOOSE

Chooses a string from a list of strings based on a given position. You specify the position and the value. If the value matches the position, the Data Integration Service returns the value.

Syntax

```
CHOOSE( index, string1 [, string2, ..., stringN] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
index	Required	Numeric datatype. Enter a number based on the position of the value you want to match.
string	Required	Any character value.

Return Value

The string that matches the position of the index value.

NULL if no string matches the position of the index value.

Example

The following expression returns the string ‘flashlight’ based on an index value of 2:

```
CHOOSE( 2, 'knife', 'flashlight', 'diving hood' )
```

The following expression returns NULL based on an index value of 4:

```
CHOOSE( 4, 'knife', 'flashlight', 'diving hood' )
```

CHOOSE returns NULL because the expression does not contain a fourth argument.

CHR

When the Data Integration Service uses ASCII mode, CHR returns the ASCII character corresponding to the numeric value you pass to this function. ASCII values fall in the range 0 to 255. You can pass any integer to CHR, but only ASCII codes 32 to 126 are printable characters.

When the Data Integration Service uses Unicode mode, CHR returns the Unicode character corresponding to the numeric value you pass to this function. Unicode values fall in the range 0 to 65,535.

Syntax

```
CHR( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. The value you want to return as an ASCII or Unicode character. You can enter any valid transformation expression.

Return Value

ASCII or Unicode character. A string containing one character.

NULL if a value passed to the function is NULL.

Example

The following expression returns the ASCII or Unicode character for each numeric value in the ITEM_ID port:

```
CHR( ITEM_ID )
```

ITEM_ID	RETURN VALUE
65	A
122	z
NULL	NULL
88	X
100	d
71	G

Use the CHR function to concatenate a single quote onto a string. The single quote is the only character that you cannot use inside a string literal. Consider the following example:

```
'Joan' || CHR(39) || 's car'
```

The return value is:

```
Joan's car
```

CHRCODE

When the Data Integration Service uses ASCII mode, CHRCODE returns the numeric ASCII value of the first character of the string passed to the function. ASCII values fall in the range 0 to 255.

When the Data Integration Service uses Unicode mode, CHRCODE returns the numeric Unicode value of the first character of the string passed to the function. Unicode values fall in the range 0 to 65,535.

Normally, before you pass any string value to CHRCODE, you parse out the specific character you want to convert to an ASCII or Unicode value. For example, you might use RTRIM or another string-manipulation function. If you pass a numeric value, CHRCODE converts it to a character string and returns the ASCII or Unicode value of the first character in the string.

This function is identical in behavior to the ASCII function. If you currently use ASCII in expressions, it will still work correctly. However, when you create new expressions, use the CHRCODE function instead of the ASCII function.

Syntax

```
CHRCODE ( string )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
string	Required	Character string. Passes the values you want to return as ASCII or Unicode values. You can enter any valid transformation expression.

Return Value

Integer.

NULL if a value passed to the function is NULL.

Example

The following expression returns the ASCII or Unicode value for the first character of each value in the ITEMS port:

```
CHRCODE( ITEMS )
```

ITEMS	RETURN VALUE
Flashlight	70
Compass	67
Safety Knife	83
Depth/Pressure Gauge	68
Regulator System	82

COLLECT_LIST

Returns an array with elements based on the argument. The data type of the argument determines the data type of the array. COLLECT_LIST is an aggregate function.

Syntax

```
COLLECT_LIST(value as any)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Any data type. The values that you want to aggregate into a hierarchical data of type array. You can enter any valid transformation expression.

Return Value

Array.

Group By

COLLECT_LIST groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, COLLECT_LIST treats all rows as one group, returning one value.

Examples

The following expression returns an array with the elements in the PRODUCT_NAME.

```
COLLECT_LIST(PRODUCT_NAME)
```

PRODUCT_NAME
Flashlight
Compass
Pressure Gauge
Vest

RETURN VALUE: [Flashlight,Compass,Pressure Gauge,Vest]

COLLECT_MAP

Returns a map with elements based on the specified arguments.

Syntax

```
COLLECT_MAP(map_key as ANY, map_value as ANY)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
map_key	Required	Any primitive data type. The elements that you want to aggregate as keys of a hierarchical data of type map. You can enter any valid transformation expression.
map_value	Required	Any primitive or complex data type. The elements that you want to aggregate as values of a hierarchical data of type map. You can enter any valid transformation expression.

Return Value

Map.

Group By

COLLECT_MAP groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, COLLECT_MAP treats all rows as one group, returning one value.

Examples

The following expression returns a map with the elements in the PRODUCT_ID as keys and elements in the PRODUCT_NAME as values.

```
COLLECT_MAP(PRODUCT_ID, PRODUCT_NAME)
```

PRODUCT_ID	PRODUCT_NAME
34890	Flashlight
12754	Compass
54028	Pressure Gauge
81203	Vest

RETURN VALUE:

```
[34890 -> Flashlight, 12754 -> Compass, 54028 -> Pressure Gauge, 81203 -> Vest]
```

COMPRESS

Compresses data using the zlib 1.2.1 compression algorithm. Use the COMPRESS function before you send large amounts of data over a wide area network.

Syntax

```
COMPRESS( value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
value	Required	String datatype. Data that you want to compress.

Return Value

Compressed binary value of the input value.

NULL if the input is a null value.

Example

Your organization has an online order service. You want to send customer order data over a wide area network. The source contains a row that is 10 MB. You can compress the data in this row using COMPRESS. When you compress the data, you decrease the amount of data the Data Integration Service writes over the network. As a result, you may increase performance.

CONCAT

Concatenates two strings. CONCAT converts all data to text before concatenating the strings. Alternatively, use the || string operator to concatenate strings. Using the || string operator instead of CONCAT improves Data Integration Service performance.

Syntax

```
CONCAT( first_string, second_string )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
first_string	Required	Any datatype except Binary. The first part of the string you want to concatenate. You can enter any valid transformation expression.
second_string	Required	Any datatype except Binary. The second part of the string you want to concatenate. You can enter any valid transformation expression.

Return Value

String.

NULL if both string values are NULL.

Nulls

If one of the strings is NULL, CONCAT ignores it and returns the other string.

If both strings are NULL, CONCAT returns NULL.

Example

The following expression concatenates the names in the FIRST_NAME and LAST_NAME ports:

```
CONCAT( FIRST_NAME, LAST_NAME )
```

FIRST_NAME	LAST_NAME	RETURN VALUE
John	Baer	JohnBaer
NULL	Campbell	Campbell
Bobbi	Apperley	BobbiApperley
Jason	Wood	JasonWood
Dan	Covington	DanCovington
Greg	NULL	Greg
NULL	NULL	NULL
100	200	100200

CONCAT does not add spaces to separate strings. If you want to add a space between two strings, you can write an expression with two nested CONCAT functions. For example, the following expression first concatenates a space on the end of the first name and then concatenates the last name:

```
CONCAT( CONCAT( FIRST_NAME, ' ' ), LAST_NAME )
```

FIRST_NAME	LAST_NAME	RETURN VALUE
John	Baer	John Baer
NULL	Campbell	Campbell *(includes leading blank)*
Bobbi	Apperley	Bobbi Apperley
Jason	Wood	Jason Wood
Dan	Covington	Dan Covington
Greg	NULL	Greg
NULL	NULL	NULL

Use the CHR and CONCAT functions to concatenate a single quote onto a string. The single quote is the only character you cannot use inside a string literal. Consider the following example:

```
CONCAT( 'Joan', CONCAT( CHR(39), 's car' ))
```

The return value is:

```
Joan's car
```

CONCAT_ARRAY

Concatenates string elements in an array based on a separator that you specify and returns a string.

Syntax

```
CONCAT_ARRAY(' ', array)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
''	Required	Each string element is separated by the separator you specify. For example, ',' separates the values with a comma.
array	Required	An array with elements of string type. The array that you want to concatenate.

Return Value

String

Nulls

If one of the string elements is NULL, CONCAT_ARRAY ignores it and returns the other string.

If all the string elements are NULL, CONCAT_ARRAY returns an empty string.

Examples

The following expression concatenates the string elements in the array.

```
CONCAT_ARRAY( ':', Name )
```

Name	RETURN VALUE
['John','Baer']	'John:Baer'
['Bobbi','Apperley']	'Bobbi:Apperley'
['Jason','']	'Jason:'
['Greg',NULL]	'Greg'
[NULL,NULL]	''

CONVERT_BASE

Converts a non-negative numeric string from one base value to another base value.

Syntax

```
CONVERT_BASE( value, source_base, dest_base )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	String datatype. Value you want to convert from one base to another base. Maximum is 9,233,372,036,854,775,806.
source_base	Required	Numeric datatype. Current base value of the data you want to convert. Minimum base is 2. Maximum base is 36.
dest_base	Required	Numeric datatype. Base value you want to convert the data to. Minimum base is 2. Maximum base is 36.

Return Value

Numeric value.

Example

The following example converts 2222 from the decimal base value 10 to the binary base value 2:

```
CONVERT_BASE( "2222", 10, 2 )
```

The Data Integration Service returns 100010101110.

COS

Returns the cosine of a numeric value (expressed in radians).

Syntax

```
COS( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the values for which you want to calculate a cosine. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the cosine for all values in the Degrees port:

```
COS( DEGREES * 3.14159265359 / 180 )
```

DEGREES	RETURN VALUE
0	1.0
90	0.0
70	0.342020143325593
30	0.866025403784421
5	0.996194698091745
18	0.951056516295147
89	0.0174524064371813
NULL	NULL

Tip: You can perform arithmetic on the values passed to COS before the function calculates the cosine. For example, you can convert the values in the port to radians before calculating the cosine, as follows:

```
COS( ARCS * 3.14159265359 / 180 )
```

COSH

Returns the hyperbolic cosine of a numeric value (expressed in radians).

Syntax

```
COSH( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the values for which you want to calculate the hyperbolic cosine. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the hyperbolic cosine for the values in the Angles port:

```
COSH( ANGLES )
```

ANGLES	RETURN VALUE
1.0	1.54308063481524
2.897	9.0874465864177
3.66	19.4435376920294
5.45	116.381231106176
0	1.0
0.345	1.06010513656773
NULL	NULL

Tip: You can perform arithmetic on the values passed to COSH before the function calculates the hyperbolic cosine. For example:

```
COSH( MEASURES.ARCS / 360 )
```

COUNT

Returns the number of rows that have non-null values in a group. Optionally, you can include the asterisk (*) argument to count all input values in a transformation. You can nest only one other aggregate function within COUNT. You can apply a condition to filter rows before counting them.

Syntax

```
COUNT( value [, filter_condition] )
```

or

```
COUNT( * [, filter_condition] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Passes the values you want to count. You can enter any valid transformation expression.
*	Optional	Use to count *all rows* in a transformation.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Integer.

0 if all values passed to this function are NULL or no rows are selected, unless you include the asterisk argument.

Nulls

If all values are NULL, the function returns 0.

If you apply the asterisk argument, this function counts all rows, regardless if a column in a row contains a null value.

If you apply the *value* argument, this function ignores columns with null values.

Group By

COUNT groups values based on group by ports you define in the transformation, returning one result for each group. If there is no group by port COUNT treats all rows as one group, returning one value.

Examples

The following expression counts the items with less than 5 quantity in stock, excluding null values:

```
COUNT( ITEM_NAME, IN_STOCK < 5 )
```

ITEM_NAME	IN_STOCK
Flashlight	10
NULL	2
Compass	NULL
Regulator System	5
Safety Knife	8
Halogen Flashlight	1

RETURN VALUE: 1

In this example, the function counted the Halogen flashlight but not the NULL item. The function counts all rows in a transformation, including null values, as illustrated in the following example:

```
COUNT( *, QTY < 5 )
```

ITEM_NAME	QTY
Flashlight	10
NULL	2
Compass	NULL
Regulator System	5
Safety Knife	8

ITEM_NAME	QTY
Halogen Flashlight	1

RETURN VALUE: 2

In this example, the function counts the NULL item and the Halogen Flashlight. If you include the asterisk argument, but do not use a filter, the function counts all rows that pass into the transformation. For example:

```
COUNT( * )
```

ITEM_NAME	QTY
Flashlight	10
NULL	2
Compass	NULL
Regulator System	5
Safety Knife	8
Halogen Flashlight	1

RETURN VALUE: 6

COUNT and Complex Data Types

You can use COUNT to count the number of rows in a complex port of type array or struct.

For example, you have the following array:

```
emp_phones =
[205-128-6478, 722-515-2889]
[107-081-0961, 718-051-8116]
[344-894-6463, 861-411-8361]
[107-031-0961, NULL]
```

You can use the following expression to count the number of rows in the array port:

```
COUNT( emp_phones )
```

RETURN VALUE: 4

CRC32

Returns a 32-bit Cyclic Redundancy Check (CRC32) value. Use CRC32 to find data transmission errors. You can also use CRC32 if you want to verify that data stored in a file has not been modified.

If you use CRC32 to perform a redundancy check on data in ASCII mode and Unicode mode, the Data Integration Service might generate different results on the same input value. If you use CRC32 to perform a redundancy check on data on different operating systems, the Data Integration Service might generate different results on the same input value.

Note: CRC32 can return the same output for different input strings. If you want to generate keys in a mapping, use a Sequence Generator transformation. If you use CRC32 to generate keys in a mapping, you might receive unexpected results.

Syntax

```
CRC32( value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
value	Required	String or Binary datatype. Passes the values you want to perform a redundancy check on. Input value is case sensitive. The case of the input value affects the return value. For example, CRC32(informatica) and CRC32 (Informatica) return different values.

Return Value

32-bit integer value.

Example

You want to read data from a source across a wide area network. You want to make sure the data has been modified during transmission. You can compute the checksum for the data in the file and store it along with the file. When you read the source data, the Data Integration Service can use CRC32 to compute the checksum and compare it to the stored value. If the two values are the same, the data has not been modified.

CREATE_TIMESTAMP_TZ

Construct a Timestamp with Time Zone data type from the timestamp and time zone values.

The output port must be timestampWithTZ for CREATE_TIMESTAMP_TZ expressions.

Syntax

```
CREATE_TIMESTAMP_TZ (timestamp_value, timezone_value)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
timestamp_value	Required	Date/Time datatype. You can enter any valid transformation expression.
timezone_value	Required	Must be a string data type. The string must be a character string. Passes the values you want to create for time zone. You can enter any valid transformation expression as defined in the time zone file present in the install location.

Return Value

Returns a timestamp with time zone data type.

NULL if the input is a null value.

Example

INPUT VALUE	RETURN VALUE
1947-08-05 10:45:00.221111000 AM, 'America/Los_Angeles'	'1947-08-05 10:45:00.221111000 AM America/Los_Angeles'
1947-08-05 10:45:00.221111000 AM, '-08:00'	'1947-08-05 10:45:00.221111000 AM -08:00'

CUME

Returns a running total. A running total means CUME returns a total each time it adds a value. You can add a condition to filter rows out of the row set before calculating the running total.

Use CUME and similar functions (such as MOVINGAVG and MOVINGSUM) to simplify reporting by calculating running values.

Syntax

```
CUME( numeric_value [, filter_condition] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values for which you want to calculate a running total. You can enter any valid transformation expression. You can create a nested expression to calculate a running total based on the results of the function as long as the result is a numeric value.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Nulls

If a value is NULL, CUME returns the running total for the previous row. However, if all values in the selected port are NULL, CUME returns NULL.

Examples

The following sample rowset might result from using the CUME function:

```
CUME( PERSONAL_SALES )
```

PERSONAL_SALES	RETURN VALUE
40000	40000
80000	120000
40000	160000
60000	220000
NULL	220000
50000	270000

Likewise, you can add values before calculating a running total:

```
CUME( CA_SALES + OR_SALES )
```

CA_SALES	OR_SALES	RETURN VALUE
40000	10000	50000
80000	50000	180000
40000	2000	222000
60000	NULL	222000
NULL	NULL	222000
50000	3000	275000

DATE_COMPARE

Returns an integer indicating which of two dates is earlier. DATE_COMPARE returns an integer value rather than a date value.

Syntax

```
DATE_COMPARE( date1, date2 )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
date1	Required	Date/Time datatype. The first date you want to compare. You can enter any valid transformation expression as long as it evaluates to a date.
date2	Required	Date/Time datatype. The second date you want to compare. You can enter any valid transformation expression as long as it evaluates to a date.

Return Value

-1 if the first date is earlier.

0 if the two dates are equal.

1 if the second date is earlier.

NULL if one of the date values is NULL.

Example

The following expression compares each date in the DATE_PROMISED and DATE_SHIPPED ports, and returns an integer indicating which date is earlier:

```
DATE_COMPARE( DATE_PROMISED, DATE_SHIPPED )
```

DATE_PROMISED	DATE_SHIPPED	RETURN VALUE
Jan 1 1997	Jan 13 1997	-1
Feb 1 1997	Feb 1 1997	0
Dec 22 1997	Dec 15 1997	1
Feb 29 1996	Apr 12 1996	-1 *(Leap year)*
NULL	Jan 6 1997	NULL
Jan 13 1997	NULL	NULL

DATE_DIFF

Returns the length of time between two dates. You can request the format to be years, months, days, hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. The Data Integration Service subtracts the second date from the first date and returns the difference.

The Data Integration Service calculates the DATE_DIFF function based on the number of months instead of the number of days. It calculates the date differences for partial months with the days selected in each month. To calculate the date difference for the partial month, the Data Integration Service adds the days used within the month. It then divides the value with the total number of days in the selected month.

The Data Integration Service gives a different value for the same period in the leap year period and a non-leap year period. The difference occurs when February is part of the DATE_DIFF function. The DATE_DIFF divides the days with 29 for February for a leap year and 28 if it is not a leap year.

For example, you want to calculate the number of months from September 13 to February 19. In a leap year period, the DATE_DIFF function calculates the month of February as 19/29 months or 0.655 months. In a non-leap year period, the DATE_DIFF function calculates the month of February as 19/28 months or 0.678 months. The Data Integration Service similarly calculates the difference in the dates for the remaining months and the DATE_DIFF function returns the totaled value for the specified period.

Note: Some databases might use a different algorithm to calculate the difference in dates.

Syntax

```
DATE_DIFF( date1, date2, format )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
date1	Required	Date/Time datatype. Passes the values for the first date you want to compare. You can enter any valid transformation expression.
date2	Required	Date/Time datatype. Passes the values for the second date you want to compare. You can enter any valid transformation expression.
format	Required	Format string specifying the date or time measurement. You can specify years, months, days, hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. You can specify only one part of the date, such as 'mm'. Enclose the format strings within single quotation marks. The format string is not case sensitive. For example, the format string 'mm' is the same as 'MM', 'Mm' or 'mM'.

Return Value

Double value. If *date1* is later than *date2*, the return value is a positive number. If *date1* is earlier than *date2*, the return value is a negative number.

0 if the dates are the same.

NULL if one (or both) of the date values is NULL.

Examples

The following expressions return the number of hours between the DATE_PROMISED and DATE_SHIPPED ports:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'HH' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'HH12' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'HH24' )
```

DATE_PROMISED	DATE_SHIPPED	RETURN VALUE
Jan 1 1997 12:00:00AM	Mar 29 1997 12:00:00PM	-2100
Mar 29 1997 12:00:00PM	Jan 1 1997 12:00:00AM	2100
NULL	Dec 10 1997 5:55:10PM	NULL
Dec 10 1997 5:55:10PM	NULL	NULL

DATE_PROMISED	DATE_SHIPPED	RETURN VALUE
Jun 3 1997 1:13:46PM	Aug 23 1996 4:20:16PM	6812.89166666667
Feb 19 2004 12:00:00PM	Feb 19 2005 12:00:00PM	-8784

The following expressions return the number of days between the DATE_PROMISED and the DATE_SHIPPED ports:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'D' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'DD' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'DDD' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'DY' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'DAY' )
```

DATE_PROMISED	DATE_SHIPPED	RETURN VALUE
Jan 1 1997 12:00:00AM	Mar 29 1997 12:00:00PM	-87.5
Mar 29 1997 12:00:00PM	Jan 1 1997 12:00:00AM	87.5
NULL	Dec 10 1997 5:55:10PM	NULL
Dec 10 1997 5:55:10PM	NULL	NULL
Jun 3 1997 1:13:46PM	Aug 23 1996 4:20:16PM	283.870486111111
Feb 19 2004 12:00:00PM	Feb 19 2005 12:00:00PM	-366

The following expressions return the number of months between the DATE_PROMISED and DATE_SHIPPED ports:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'MM' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'MON' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'MONTH' )
```

DATE_PROMISED	DATE_SHIPPED	RETURN VALUE
Jan 1 1997 12:00:00AM	Mar 29 1997 12:00:00PM	-2.91935483870968
Mar 29 1997 12:00:00PM	Jan 1 1997 12:00:00AM	2.91935483870968
NULL	Dec 10 1997 5:55:10PM	NULL
Dec 10 1997 5:55:10PM	NULL	NULL
Jun 3 1997 1:13:46PM	Aug 23 1996 4:20:16PM	9.3290162037037
Feb 19 2004 12:00:00PM	Feb 19 2005 12:00:00PM	-12

The following expressions return the number of years between the DATE_PROMISED and DATE_SHIPPED ports:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'Y' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'YY' )
```

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'YYY' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'YYYY' )
```

DATE_PROMISED	DATE_SHIPPED	RETURN VALUE
Jan 1 1997 12:00:00AM	Mar 29 1997 12:00:00PM	-0.24327956989247
Mar 29 1997 12:00:00PM	Jan 1 1997 12:00:00AM	0.24327956989247
NULL	Dec 10 1997 5:55:10PM	NULL
Dec 10 1997 5:55:10PM	NULL	NULL
Jun 3 1997 1:13:46PM	Aug 23 1996 4:20:16PM	0.77741801697531
Feb 19 2004 12:00:00PM	Feb 19 2005 12:00:00PM	-1

The following expressions return the number of months between the DATE_PROMISED and DATE_SHIPPED ports:

```
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'MM' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'MON' )
DATE_DIFF( DATE_PROMISED, DATE_SHIPPED, 'MONTH' )
```

DATE_PROMISED	DATE_SHIPPED	LEAP YEAR VALUE (in Months)	NON-LEAP YEAR VALUE (in Months)
Sept 13	Feb 19	-5.237931034	-5.260714286
NULL	Feb 19	NULL	N/A
Sept 13	NULL	NULL	N/A

DEC_BASE64

Decodes a base 64 encoded value and returns a string with the binary data representation of the data. If you encode data using ENC_BASE64, and you want to decode data using DEC_BASE64, you must run the mapping using the same data movement mode. Otherwise, the output of the decoded data may differ from the original data.

Syntax

```
DEC_BASE64( value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
value	Required	String datatype. Data that you want to decode.

Return Value

Binary decoded value.

NULL if the input is a null value.

Return values differ if the mapping runs in Unicode mode versus ASCII mode.

DECODE

Searches a port for a value you specify. If the function finds the value, it returns a result value, which you define. You can build an unlimited number of searches within a DECODE function.

If you use DECODE to search for a value in a string port, you can either trim trailing blanks with the RTRIM function or include the blanks in the search string.

Syntax

```
DECODE( value, first_search, first_result [, second_search, second_result]...[,default] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Passes the values you want to search. You can enter any valid transformation expression.
search	Required	Any value with the same datatype as the value argument. Passes the values for which you want to search. The search value must match the value argument. You cannot search for a portion of a value. Also, the search value is case sensitive. For example, if you want to search for the string 'Halogen Flashlight' in a particular port, you must enter 'Halogen Flashlight, not just 'Halogen'. If you enter 'Halogen', the search does not find a matching value. You can enter any valid transformation expression.
result	Required	Any datatype except Binary. The value you want to return if the search finds a matching value. You can enter any valid transformation expression.
default	Optional	Any datatype except Binary. The value you want to return if the search does not find a matching value. You can enter any valid transformation expression.

Return Value

First_result if the search finds a matching value.

Default value if the search does not find a matching value.

NULL if you omit the default argument and the search does not find a matching value.

Even if multiple conditions are met, the Data Integration Service returns the first matching result.

If the data contains multibyte characters and the DECODE expression compares string data, the return value depends on the code page and data movement mode of the Data Integration Service.

DECODE and Datatypes

When you use DECODE, the datatype of the return value is always the same as the datatype of the result with the greatest precision.

For example, you have the following expression:

```
DECODE ( CONST_NAME
         'Five', 5,
         'Pythagoras', 1.414213562,
         'Archimedes', 3.141592654,
         'Pi', 3.141592654 )
```

The return values in this expression are 5, 1.414213562, and 3.141592654. The first result is an Integer, and the other results are Decimal. The Decimal datatype has greater precision than Integer. This expression always writes the result as a Decimal.

When you run a mapping in high precision mode, if at least one result is Double, the datatype of the return value is Double.

You cannot create a DECODE function with both string and numeric return values.

For example, the following expression is invalid:

```
DECODE ( CONST_NAME
         'Five', 5,
         'Pythagoras', '1.414213562',
         'Archimedes', '3.141592654',
         'Pi', 3.141592654 )
```

When you validate the expression above, you receive the following error message:

```
Function cannot resolve operands of ambiguously mismatching datatypes.
```

Examples

You might use DECODE in an expression that searches for a particular ITEM_ID and returns the ITEM_NAME:

```
DECODE( ITEM_ID, 10, 'Flashlight',
                 14, 'Regulator',
                 20, 'Knife',
                 40, 'Tank',
                 'NONE' )
```

ITEM_ID	RETURN VALUE
10	Flashlight
14	Regulator
17	NONE
20	Knife
25	NONE
NULL	NONE
40	Tank

DECODE returns the default value of NONE for items 17 and 25 because the search values did not match the ITEM_ID. Also, DECODE returns NONE for the NULL ITEM_ID.

The following expression tests multiple columns and conditions, evaluated in a top to bottom order for TRUE or FALSE:

```
DECODE( TRUE,
        Var1 = 22, 'Variable 1 was 22!',
        Var2 = 49, 'Variable 2 was 49!',
        Var1 < 23, 'Variable 1 was less than 23.',
```

```
            Var2 > 30, 'Variable 2 was more than 30.',
            'Variables were out of desired ranges.')
```

Var1	Var2	RETURN VALUE
21	47	Variable 1 was less than 23.
22	49	Variable 1 was 22!
23	49	Variable 2 was 49!
24	27	Variables were out of desired ranges.
25	50	Variable 2 was more than 30.

DECOMPRESS

Decompresses data using the zlib 1.2.1 compression algorithm. Use the DECOMPRESS function on data that has been compressed with the COMPRESS function or a compression tool that uses the zlib 1.2.1 algorithm. If the mapping that decompresses the data uses a different data movement mode than the mapping that compressed the data, the output of the decompressed data may differ from the original data.

Syntax

```
DECOMPRESS( value, precision )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Binary datatype. Data that you want to decompress.
precision	Optional	Integer datatype.

Return Value

Decompressed binary value of the input value.

NULL if the input is a null value.

ENC_BASE64

Encodes data by converting binary data to string data using Multipurpose Internet Mail Extensions (MIME) encoding. Encode data when you want to store data in a database or file that does not allow binary data. You can also encode data to pass binary data through transformations in string format. The encoded data is approximately 33% longer than the original data. It displays as a set of random characters.

Syntax

```
ENC_BASE64( value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
value	Required	Binary or String datatype. Data that you want to encode.

Return Value

Encoded value.

NULL if the input is a null value.

ERROR

Causes the Data Integration Service to skip a row and issue an error message, which you define. The error message displays in the log. The Data Integration Service does not write these skipped rows to the reject file.

Use ERROR in Expression transformations to validate data. Generally, you use ERROR within an IIF or DECODE function to set rules for skipping rows.

Use the ERROR function for both input and output port default values. You might use ERROR for input ports to keep null values from passing into a transformation.

Use ERROR for output ports to handle any kind of transformation error, including ERROR function calls within an expression. When you use the ERROR function in an expression and in the output port default value, the Data Integration Service skips the row and logs both the error message from the expression and the error message from the default value. If you want to ensure the Data Integration Service skips rows that produce an error, assign ERROR as the default value.

If you use an output default value other than ERROR, the default value overrides the ERROR function in an expression. For example, you use the ERROR function in an expression, and you assign the default value, '1234', to the output port. Each time the Data Integration Service encounters the ERROR function in the expression, it overrides the error with the value '1234' and passes '1234' to the next transformation. It does not skip the row, and it does not log an error in the log.

Syntax

```
ERROR( string )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
string	Required	String value. The message you want to display when the Integration Service skips a row based on the expression containing the ERROR function. The string can be any length.

Return Value

String.

Example

The following example shows how to reference a mapping that calculates the average salary for employees in all departments of the organization, but skip negative values. The following expression nests the ERROR function in an IIF expression so that if the Data Integration Service finds a negative salary in the Salary port, it skips the row and displays an error:

```
IIF( SALARY < 0, ERROR ('Error. Negative salary found. Row skipped.', EMP_SALARY )
```

SALARY	RETURN VALUE
10000	10000
-15000	'Error. Negative salary found. Row skipped.'
NULL	NULL
150000	150000
1005	1005

EXP

Returns e raised to the specified power (exponent), where e=2.71828183. For example, EXP(2) returns 7.38905609893065. You might use this function to analyze scientific and technical data rather than business data. EXP is the reciprocal of the LN function, which returns the natural logarithm of a numeric value.

Syntax

```
EXP( exponent )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
exponent	Required	Numeric datatype. The value to which you want to raise e. The exponent in the equation e^*value*. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed as an argument to the function is NULL.

Example

The following expression uses the values stored in the Numbers port as the exponent value:

```
EXP( NUMBERS )
```

NUMBERS	RETURN VALUE
10	22026.4657948067

NUMBERS	RETURN VALUE
-2	0.135335283236613
8.55	5166.754427176
NULL	NULL

EXTRACT_STRUCT

Extracts all elements from a dynamic struct port in an Expression transformation. Use the dot operator in expressions to extract elements of a struct. For more information on using the dot operator, see the **Data Engineering Integration User Guide**.

When you use the EXTRACT_STRUCT function, you must specify a base port in which you set the dynamic struct field that is being used as the argument.

You cannot nest other functions within the EXTRACT_STRUCT function or nest the EXTRACT_STRUCT function in another function.

Syntax

```
EXTRACT_STRUCT(dynamic_struct1)
```

The following table describes the arguments for this command:

Argument	Required/Optional	Description
Dynamic struct	Required	Extracts all elements from a dynamic struct port in an Expression transformation.

Return Value

Dynamic type consisting of all the struct elements as generated fields.

Example

The following function extracts and flattens a dynamic struct:

```
EXTRACT_STRUCT(Struct_emp_info_struct_as)
```

FIRST

Returns the first value found within a port or group. Optionally, you can apply a filter to limit the rows the Data Integration Service reads. You can nest only one other aggregate function within FIRST.

Syntax

```
FIRST( value [, filter_condition ] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Passes the values for which you want to return the first value. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

First value in a group.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Nulls

If a value is NULL, FIRST ignores the row. However, if all values passed from the port are NULL, FIRST returns NULL.

Group By

FIRST groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, FIRST treats all rows as one group, returning one value.

Examples

The following expression returns the first value in the ITEM_NAME port with a price greater than $10.00:

```
FIRST( ITEM_NAME, ITEM_PRICE > 10 )
```

ITEM_NAME	ITEM_PRICE
Flashlight	35.00
Navigation Compass	8.05
Regulator System	150.00
Flashlight	29.00
Depth/Pressure Gauge	88.00
Flashlight	31.00

RETURN VALUE: Flashlight

The following expression returns the first value in the ITEM_NAME port with a price greater than $40.00:

```
FIRST( ITEM_NAME, ITEM_PRICE > 40 )
```

ITEM_NAME	ITEM_PRICE
Flashlight	35.00
Navigation Compass	8.05
Regulator System	150.00
Flashlight	29.00
Depth/Pressure Gauge	88.00
Flashlight	31.00

RETURN VALUE: Regulator System

FLOOR

Returns the largest integer less than or equal to the numeric value you pass to this function. For example, if you pass 3.14 to FLOOR, the function returns 3. If you pass 3.98 to FLOOR, the function returns 3. Likewise, if you pass -3.17 to FLOOR, the function returns -4.

Syntax

```
FLOOR( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. You can enter any valid transformation expression as long as it evaluates to numeric data.

Return Value

Integer if you pass a numeric value with declared precision between 0 and 28.

Double if you pass a numeric value with declared precision greater than 28.

NULL if a value passed to the function is NULL.

Example

The following expression returns the largest integer less than or equal to the values in the PRICE port:

```
FLOOR( PRICE )
```

PRICE	RETURN VALUE
39.79	39

PRICE	RETURN VALUE
125.12	125
74.24	74
NULL	NULL
-100.99	-101

Tip: You can perform arithmetic on the values you pass to FLOOR. For example, to multiply a numeric value by 10 and then calculate the largest integer that is less than the product, you might write the function as follows:

```
FLOOR( UNIT_PRICE * 10 )
```

FV

Returns the future value of an investment, where you make periodic, constant payments and the investment earns a constant interest rate.

Syntax

```
FV( rate, terms, payment [, present value, type] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
rate	Required	Numeric. Interest rate earned in each period. Expressed as a decimal number. Divide the percent rate by 100 to express it as a decimal number.
terms	Required	Numeric. Number of periods or payments. Must be greater than 0. **Note:** The Spark engine writes null values for rows when the terms argument passes a 0 value. In the native environment, the Data Integration Service rejects the row and does not write it to the target.
payment	Required	Numeric. Payment amount due per period. Must be a negative number
present value	Optional	Numeric. Current value of the investment. If you omit this argument, FV uses 0.
type	Optional	Integer. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, the Data Integration Service treats the value as 1.

Return Value

Numeric.

Example

You deposit $2,000 into an account that earns 9% annual interest compounded monthly (monthly interest of 9%/12, or 0.75%). You plan to deposit $250 at the beginning of every month for the next 12 months. The following expression returns $5,337.96 as the account balance at the end of 12 months:

```
FV(0.0075, 12, -250, -2000, TRUE)
```

Notes

To calculate interest rate earned in each period, divide the annual rate by the number of payments made in a year. The payment value and present value are negative because these are amounts that you pay.

GET_DATE_PART

Returns the specified part of a date as an integer value. Therefore, if you create an expression that returns the month portion of the date, and pass a date such as Apr 1 1997 00:00:00, GET_DATE_PART returns 4.

Syntax

```
GET_DATE_PART( date, format )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. You can enter any valid transformation expression.
format	Required	A format string specifying the portion of the date value you want to return. Enclose format strings within single quotation marks, for example, 'mm'. The format string is not case sensitive. Each format string returns the entire part of the date based on the date format specified in the mapping. For example, if you pass the date Apr 1 1997 to GET_DATE_PART, the format strings 'Y', 'YY', 'YYY', or 'YYYY' all return 1997.

Return Value

Integer representing the specified part of the date.

NULL if a value passed to the function is NULL.

Examples

The following expressions return the hour for each date in the DATE_SHIPPED port. 12:00:00AM returns 0 because the default date format is based on the 24 hour interval:

```
GET_DATE_PART( DATE_SHIPPED, 'HH' )
GET_DATE_PART( DATE_SHIPPED, 'HH12' )
GET_DATE_PART( DATE_SHIPPED, 'HH24' )
```

DATE_SHIPPED	RETURN VALUE
Mar 13 1997 12:00:00AM	0
Sep 2 1997 2:00:01AM	2

DATE_SHIPPED	RETURN VALUE
Aug 22 1997 12:00:00PM	12
June 3 1997 11:30:44PM	23
NULL	NULL

The following expressions return the day for each date in the DATE_SHIPPED port:

```
GET_DATE_PART( DATE_SHIPPED, 'D' )
GET_DATE_PART( DATE_SHIPPED, 'DD' )
GET_DATE_PART( DATE_SHIPPED, 'DDD' )
GET_DATE_PART( DATE_SHIPPED, 'DY' )
GET_DATE_PART( DATE_SHIPPED, 'DAY' )
```

DATE_SHIPPED	RETURN VALUE
Mar 13 1997 12:00:00AM	13
June 3 1997 11:30:44PM	3
Aug 22 1997 12:00:00PM	22
NULL	NULL

The following expressions return the month for each date in the DATE_SHIPPED port:

```
GET_DATE_PART( DATE_SHIPPED, 'MM' )
GET_DATE_PART( DATE_SHIPPED, 'MON' )
GET_DATE_PART( DATE_SHIPPED, 'MONTH' )
```

DATE_SHIPPED	RETURN VALUE
Mar 13 1997 12:00:00AM	3
June 3 1997 11:30:44PM	6
NULL	NULL

The following expression return the year for each date in the DATE_SHIPPED port:

```
GET_DATE_PART( DATE_SHIPPED, 'Y' )
GET_DATE_PART( DATE_SHIPPED, 'YY' )
GET_DATE_PART( DATE_SHIPPED, 'YYY' )
GET_DATE_PART( DATE_SHIPPED, 'YYYY' )
```

DATE_SHIPPED	RETURN VALUE
Mar 13 1997 12:00:00AM	1997
June 3 1997 11:30:44PM	1997
NULL	NULL

GET_TIMEZONE

Returns the time zone value from a given timestamp with time zone column.

For example:

```
String TimeZone = GET_TIMEZONE (timestampWithTZ);
```

The output port must be of the String data type for the GET_TIMEZONE expressions.

Syntax

```
GET_TIMEZONE (timestamp_with_timezone_value)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
timestamp_with_timezone_value	Required	Must be a timestamp with time zone data type. You can enter any valid transformation expression.

Return Value

Returns a String data type containing the time zone region name or a time zone offset.

NULL if the input is a null value.

Example

INPUT VALUE	RETURN VALUE
‘1947-08-05 10:45:00.221111111 AM America/Los_Angeles’	‘AMERICA/LOS_ANGELES'
'1947-08-05 10:45:00.221111111 AM -08:00'	‘-08:00'

GET_TIMESTAMP

Returns the date/time value for a timestampWithTZ input type. The date/time returned will be in the requested time zone, which can be provided as the second argument. If the time zone value is not specified in the second argument, the function returns the timestamp part of the input timestampWithTZ value.

For example:

```
GET_TIMESTAMP(Timestamp with Time Zone,“+08:30”)
```

The first argument, timestamp with time zone has (+05:30) as the time zone value. The function returns the timestamp in the time zone specified as the second argument, (+08:30).

The output port must be date/time for GET_TIMESTAMP expressions.

Syntax

```
GET_TIMESTAMP (timestamp_with_timezone_value, [timezone_value])
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
timestamp_with_timezone_value	Required	Must be a timestamp with time zone data type. You can enter any valid transformation expression.
timezone_value	Optional	Must be a string data type. The string must be a character string. Passes the values you want to display for time zone based on which the function can return the timestamp. You can enter any valid transformation expression. If you do not specify the time zone, the function returns the timestamp part of the first argument.

Return Value

Returns the timestamp value in time zone offset or region specified.

If the time zone value is not passed, the function returns the timestamp part of the first argument.

NULL if the input is a null value.

Example

INPUT VALUE	RETURN VALUE
'1996-01-05 10:45:00.221111111 AM America/Los_Angeles', '+05:30'	Returns the timestamp value in time zone offset of '+05:30': '1996-01-06 12:15:00.221111111 AM'
'1996-01-05 10:45:00.221111111 AM America/Los_Angeles', 'GMT'	Returns the timestamp value in the 'GMT' time zone: '1996-01-05 06:45:00.221111111 PM'
'1996-01-05 10:45:00.221111111 AM America/Los_Angeles'	As the time zone value is not passed as the second input parameter, the timestamp is returned: '1996-01-05 10:45:00.221111111 AM'

GREATEST

Returns the greatest value from a list of input values. Use this function to return the greatest string, date, or number. By default, the match is case sensitive.

Syntax

```
GREATEST( value1, [value2, ..., valueN,] CaseFlag )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Any data type except Binary. Data type must be compatible with other values. Value you want to compare against other values. You must enter at least one value argument. If the value is numeric, and other input values are numeric, all values use the highest precision possible. For example, if some values are Integer data type and others are Double data type, the Data Integration Service converts the values to Double.
CaseFlag	Optional	Must be an integer. Specify a value when the input value argument is a string value. Determines whether the arguments in this function are case sensitive. You can enter any valid transformation expression. When CaseFlag is a number other than 0, the function is case sensitive. When CaseFlag is 0, the function is not case sensitive. Default is case sensitive.

Return Value

value1 if it is the greatest of the input values, *value2* if it is the greatest of the input values, and so on.

NULL if any of the arguments is null.

Example

The following expression returns the greatest quantity of items ordered:

```
GREATEST( QUANTITY1, QUANTITY2, QUANTITY3 )
```

QUANTITIY1	QUANTITY2	QUANTITY3	RETURN VALUE
150	756	27	756
			NULL
5000	97	17	5000
120	1724	965	1724

IIF

Returns one of two values you specify, based on the results of a condition.

Syntax

```
IIF( condition, value1 [,value2] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
condition	Required	The condition you want to evaluate. You can enter any valid transformation expression that evaluates to TRUE or FALSE.
value1	Required	Any datatype except Binary. The value you want to return if the condition is TRUE. The return value is always the datatype specified by this argument. You can enter any valid transformation expression, including another IIF expression.
value2	Optional	Any datatype except Binary. The value you want to return if the condition is FALSE. You can enter any valid transformation expression, including another IIF expression.

Unlike conditional functions in some systems, the FALSE (*value2*) condition in the IIF function is not required. If you omit *value2*, the function returns the following when the condition is FALSE:

- 0 if *value1* is a Numeric datatype.
- Empty string if *value1* is a String datatype.
- NULL if *value1* is a Date/Time datatype.

For example, the following expression does not include a FALSE condition and *value1* is a string datatype so the Data Integration Service returns an empty string for each row that evaluates to FALSE:

```
IIF( SALES > 100, EMP_NAME )
```

SALES	EMP_NAME	RETURN VALUE
150	John Smith	John Smith
50	Pierre Bleu	'' *(empty string)*
120	Sally Green	Sally Green
NULL	Greg Jones	'' *(empty string)*

Return Value

value1 if the condition is TRUE.

value2 if the condition is FALSE.

For example, the following expression includes the FALSE condition NULL so the Data Integration Service returns NULL for each row that evaluates to FALSE:

```
IIF( SALES > 100, EMP_NAME, NULL )
```

SALES	EMP_NAME	RETURN VALUE
150	John Smith	John Smith
50	Pierre Bleu	NULL

SALES	EMP_NAME	RETURN VALUE
120	Sally Green	Sally Green
NULL	Greg Jones	NULL

If the data contains multibyte characters and the condition argument compares string data, the return value depends on the code page and data movement mode of the Data Integration Service.

IIF and Datatypes

When you use IIF, the datatype of the return value is the same as the datatype of the result with the greatest precision.

For example, you have the following expression:

```
IIF( SALES < 100, 1, .3333 )
```

The TRUE result (1) is an integer and the FALSE result (.3333) is a decimal. The Decimal datatype has greater precision than Integer, so the datatype of the return value is always a Decimal.

When you run a mapping in high precision mode and at least one result is Double, the datatype of the return value is Double.

IIF and Complex Data Types

You can use IIF to return an array or a struct, or elements from the array or struct.

For example, you have the following array:

```
names = ['John', 'Kevin', 'Laura']
```

You can use the following expression to return one of the values in the array:

```
IIF( SIZE(names) > 2, names[2], names[0] )
```

RETURN VALUE: 'Laura'

Special Uses of IIF

Use nested IIF statements to test multiple conditions. The following example tests for various conditions and returns 0 if sales is 0 or negative:

```
IIF( SALES > 0, IIF( SALES < 50, SALARY1, IIF( SALES < 100, SALARY2, IIF( SALES < 200,
SALARY3, BONUS))), 0 )
```

You can make this logic more readable by adding comments:

```
IIF( SALES > 0,
--then test to see if sales is between 1 and 49:
   IIF( SALES < 50,

   --then return SALARY1
      SALARY1,

      --else test to see if sales is between 50 and 99:
         IIF( SALES < 100,

          --then return
             SALARY2,

             --else test to see if sales is between 100 and 199:
                IIF( SALES < 200,

                --then return
                   SALARY3,
```

```
                    --else for sales over 199, return
                       BONUS)
                )
             ),

--else for sales less than or equal to zero, return
      0)
```

Use IIF in update strategies. For example:

```
IIF( ISNULL( ITEM_NAME ), DD_REJECT, DD_INSERT)
```

Alternative to IIF

Use "DECODE" on page 88 instead of IIF in many cases. DECODE may improve readability. The following shows how you use DECODE instead of IIF using the first example from the previous section:

```
DECODE( TRUE,
   SALES > 0 and SALES < 50, SALARY1,
   SALES > 49 AND SALES < 100, SALARY2,
   SALES > 99 AND SALES < 200, SALARY3,
   SALES > 199, BONUS)
```

You can often use a Filter transformation instead of IIF to maximize performance.

IN

Matches input data to a list of values. By default, the match is case sensitive.

Syntax

```
IN( valueToSearch, value1, [value2, ..., valueN,] CaseFlag )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
valueToSearch	Required	Can be a string, date, or numeric value. Input value you want to match against a comma-separated list of values.
value	Required	Can be a string, date, or numeric value depending on the type specified for the valueToSearch argument. Comma-separated list of values you want to search for. Values can be ports in a transformation. There is no maximum number of values you can list.
CaseFlag	Optional	Must be an integer. Specify a value when the valueToSearch argument is a string value. Determines whether the arguments in this function are case sensitive. You can enter any valid transformation expression. When CaseFlag is a number other than 0, the function is case sensitive. When CaseFlag is 0, the function is not case sensitive. Default is case sensitive.

Return Value

TRUE (1) if the input value matches the list of values.

FALSE (0) if the input value does not match the list of values.

NULL if the input is a null value.

Example

The following expression determines if the input value is a safety knife, chisel point knife, or medium titanium knife. The input values do not have to match the case of the values in the comma-separated list:

```
IN( ITEM_NAME, 'Chisel Point Knife', 'Medium Titanium Knife', 'Safety Knife', 0 )
```

ITEM_NAME	RETURN VALUE
Stabilizing Vest	0 (FALSE)
Safety knife	1 (TRUE)
Medium Titanium knife	1 (TRUE)
	NULL

INDEXOF

Finds the index of a value among a list of values. By default, the match is case sensitive.

Syntax

```
INDEXOF( valueToSearch, string1 [, string2, ..., stringN,] [CaseFlag] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
valueToSearch	Required	String datatype. Value you want to search for in the list of strings.
string	Required	String datatype. Comma-separated list of values you want to search against. Values can be in string format. There is no maximum number of values you can list. The value is case sensitive, unless you set CaseFlag to 0.
CaseFlag	Optional	Must be an integer. Specify a value when the valueToSearch argument is a string value. Determines whether the arguments in this function are case sensitive. You can enter any valid transformation expression. When CaseFlag is a number other than 0, the function is case sensitive. When CaseFlag is 0, the function is not case sensitive.

Return Value

1 if the input value matches *string1*, 2 if the input value matches *string2*, and so on.

0 if the input value is not found.

NULL if the input is a null value.

Example

The following expression determines if values from the ITEM_NAME port match the first, second, or third string:

```
INDEXOF( ITEM_NAME, 'diving hood', 'flashlight', 'safety knife')
```

ITEM_NAME	RETURN VALUE
Safety Knife	0
diving hood	1
Compass	0
safety knife	3
flashlight	2

Safety Knife returns a value of 0 because it does not match the case of the input value.

INITCAP

Capitalizes the first letter in each word of a string and converts all other letters to lowercase. Words are delimited by white space (a blank space, formfeed, newline, carriage return, tab, or vertical tab) and characters that are not alphanumeric. For example, if you pass the string '…THOMAS', the function returns Thomas.

Syntax

```
INITCAP( string )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
string	Required	Any datatype except Binary. You can enter any valid transformation expression.

Return Value

String. If the data contains multibyte characters, the return value depends on the code page and data movement mode of the Data Integration Service.

NULL if a value passed to the function is NULL.

Example

The following expression capitalizes all names in the FIRST_NAME port:

```
INITCAP( FIRST_NAME )
```

FIRST_NAME	RETURN VALUE
ramona	Ramona
18-albert	18-Albert
NULL	NULL
?!SAM	?!Sam
THOMAS	Thomas
PierRe	Pierre

INSTR

Returns the position of a character set in a string, counting from left to right.

Syntax

```
INSTR( string, search_value [,start [,occurrence [,comparison_type ]]] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
string	Required	The string must be a character string. Passes the value you want to evaluate. You can enter any valid transformation expression. The results of the expression must be a character string. If not, INSTR converts the value to a string before evaluating it.
search_value	Required	Any value. The search value is case sensitive. The set of characters you want to search for. The search_value must match a part of the string. For example, if you write `INSTR('Alfred Pope', 'Alfred Smith')` the function returns 0. You can enter any valid transformation expression. If you want to search for a character string, enclose the characters you want to search for in single quotation marks, for example 'abc'.
start	Optional	Must be an integer value. The position in the string where you want to start the search. You can enter any valid transformation expression. The default is 1, meaning that INSTR starts the search at the first character in the string. If the start position is 0, INSTR searches from the first character in the string. If the start position is a positive number, INSTR locates the start position by counting from the beginning of the string. If the start position is a negative number, INSTR locates the start position by counting from the end of the string. If you omit this argument, the function uses the default value of 1.

Argument	Required/ Optional	Description
occurrence	Optional	A positive integer greater than 0. You can enter any valid transformation expression. If the search value appears more than once in the string, you can specify which occurrence you want to search for. For example, you would enter 2 to search for the second occurrence from the start position. If you omit this argument, the function uses the default value of 1, meaning that INSTR searches for the first occurrence of the search value. If you pass a decimal, the Data Integration Service rounds it to the nearest integer value. If you pass a negative integer or 0, the session fails.
comparison_type	Optional	The string comparison type, either linguistic or binary, when the Data Integration Service runs in Unicode mode. When the Data Integration Service runs in ASCII mode, the comparison type is always binary. Linguistic comparisons take language-specific collation rules into account, while binary comparisons perform bitwise matching. For example, the German sharp s character matches the string “ss” in a linguistic comparison, but not in a binary comparison. Binary comparisons run faster than linguistic comparisons. Must be an integer value, either 0 or 1: - 0: INSTR performs a linguistic string comparison. - 1: INSTR performs a binary string comparison. Default is 0.

Return Value

Integer if the search is successful. Integer represents the position of the first character in the *search_value*, counting from left to right.

0 if the search is unsuccessful.

NULL if a value passed to the function is NULL.

Examples

The following expression returns the position of the first occurrence of the letter ‘a’, starting at the beginning of each company name. Because the *search_value* argument is case sensitive, it skips the ‘A’ in ‘Blue Fin Aqua Center’, and returns the position for the ‘a’ in ‘Aqua’:

```
INSTR( COMPANY, 'a' )
```

COMPANY	RETURN VALUE
Blue Fin Aqua Center	13
Maco Shark Shop	2
Scuba Gear	5
Frank's Dive Shop	3
VIP Diving Club	0

The following expression returns the position of the second occurrence of the letter ‘a’, starting at the beginning of each company name. Because the *search_value* argument is case sensitive, it skips the ‘A’ in ‘Blue Fin Aqua Center’, and returns 0:

```
INSTR( COMPANY, 'a', 1, 2 )
```

COMPANY	RETURN VALUE
Blue Fin Aqua Center	0
Maco Shark Shop	8
Scuba Gear	9
Frank's Dive Shop	0
VIP Diving Club	0

The following expression returns the position of the second occurrence of the letter ‘a’ in each company name, starting from the last character in the company name. Because the *search_value* argument is case sensitive, it skips the ‘A’ in 'Blue Fin Aqua Center’, and returns 0:

```
INSTR( COMPANY, 'a', -1, 2 )
```

COMPANY	RETURN VALUE
Blue Fin Aqua Center	0
Maco Shark Shop	2
Scuba Gear	5
Frank's Dive Shop	0
VIP Diving Club	0

The following expression returns the position of the first character in the string ‘Blue Fin Aqua Center’ (starting from the last character in the company name):

```
INSTR( COMPANY, 'Blue Fin Aqua Center', -1, 1 )
```

COMPANY	RETURN VALUE
Blue Fin Aqua Center	1
Maco Shark Shop	0
Scuba Gear	0
Frank's Dive Shop	0
VIP Diving Club	0

Using Nested INSTR

You can nest the INSTR function within other functions to accomplish more complex tasks.

The following expression evaluates a string, starting from the end of the string. The expression finds the last (rightmost) space in the string and then returns all characters to the left of it:

```
SUBSTR( CUST_NAME,1,INSTR( CUST_NAME,' ' ,-1,1 ))
```

CUST_NAME	RETURN VALUE
PATRICIA JONES	PATRICIA
MARY ELLEN SHAH	MARY ELLEN

The following expression removes the character '#' from a string:

```
SUBSTR( CUST_ID, 1, INSTR(CUST_ID, '#')-1 ) || SUBSTR( CUST_ID, INSTR(CUST_ID, '#')+1 )
```

CUST_ID	RETURN VALUE
ID#33	ID33
#A3577	A3577
SS #712403399	SS 712403399

ISNULL

Returns whether a value is NULL. ISNULL evaluates an empty string as FALSE.

Note: To test for empty strings, use LENGTH.

Syntax

```
ISNULL( value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Passes the rows you want to evaluate. You can enter any valid transformation expression.

Return Value

TRUE (1) if the value is NULL.

FALSE (0) if the value is not NULL.

Example

The following example checks for null values in the items table:

```
ISNULL( ITEM_NAME )
```

ITEM_NAME	RETURN VALUE
Flashlight	0 (FALSE)
NULL	1 (TRUE)
Regulator system	0 (FALSE)
''	0 (FALSE) *Empty string is not NULL*

ISNULL and Complex Data Types

You can use ISNULL to check whether an array or a struct has a null value.

The following expressions check for the null values in the following complex data types:

Complex Data Type	Input Value	RETURN VALUE
NULL_array = NULL	ISNULL(NULL_array)	1 (TRUE)
NULL_struct = NULL	ISNULL(NULL_struct)	1 (TRUE)
num_array = [1, 2, 3]	ISNULL(num_array)	0 (FALSE)
num_array = [1, NULL, 3]	ISNULL(num_array)	0 (FALSE)

num_struct{
number: int
rank: int
}
	ISNULL(num_struct)	0 (FALSE)

IS_DATE

Returns whether a string value is a valid date. A valid date is any string in the date portion of the date time format specified in the session. If the string you want to test is not in this date format, use the TO_DATE format string to specify the date format. If the strings passed to IS_DATE do not match the format string specified, the function returns FALSE (0). If the strings match the format string, the function returns TRUE (1).

IS_DATE evaluates strings and returns an integer value.

The output port for an IS_DATE expression must be String or Numeric datatype.

You might use IS_DATE to test or filter data in a flat file before writing it to a target.

Use the RR format string with IS_DATE instead of the YY format string. In most cases, the two format strings return the same values, but there are some unique cases where YY returns incorrect results. For example, the expression IS_DATE('02/29/00', 'YY') is internally computed as IS_DATE(02/29/1900 00:00:00), which returns false. However, the Data Integration Service computes the expression IS_DATE('02/29/00', 'RR') as IS_DATE(02/29/2000 00:00:00), which returns TRUE. In the first case, year 1900 is not a leap year, so there is no February 29th.

Note: IS_DATE uses the same format strings as TO_DATE.

Syntax

```
IS_DATE( value [,format] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Must be a string datatype. Passes the rows you want to evaluate. You can enter any valid transformation expression.
format	Optional	Enter a valid TO_DATE format string. The format string must match the parts of the *string* argument. For example, if you pass the string 'Mar 15 1997 12:43:10AM', you must use the format string 'MON DD YYYY HH12:MI:SSAM'. If you omit the format string, the string value must be in the date format specified in the mapping configuration.

Return Value

TRUE (1) if the row is a valid date.

FALSE (0) if the row is not a valid date.

NULL if a value in the expression is NULL or if the format string is NULL.

Warning: The format of the IS_DATE string must match the format string, including any date separators. If it does not, the Data Integration Service might return inaccurate values or skip the record.

Examples

The following expression checks the INVOICE_DATE port for valid dates:

```
IS_DATE( INVOICE_DATE )
```

This expression returns data similar to the following:

INVOICE_DATE	RETURN VALUE
NULL	NULL
'180'	0 (FALSE)
'04/01/98'	0 (FALSE)
'04/01/1998 00:12:15.7008'	1 (TRUE)
'02/31/1998 12:13:55.9204'	0 (FALSE) (*February does not have 31 days*)
'John Smith'	0 (FALSE)

The following IS_DATE expression specifies a format string of 'YYYY/MM/DD':

```
IS_DATE( INVOICE_DATE, 'YYYY/MM/DD' )
```

If the string value does not match this format, IS_DATE returns FALSE:

INVOICE_DATE	RETURN VALUE
NULL	NULL
'180'	0 (FALSE)
'04/01/98'	0 (FALSE)
'1998/01/12'	1 (TRUE)
'1998/11/21 00:00:13'	0 (FALSE)
'1998/02/31'	0 (FALSE) (*February does not have 31 days*)
'John Smith'	0 (FALSE)

The following example shows how you use IS_DATE to test data before using TO_DATE to convert the strings to dates. This expression checks the values in the INVOICE_DATE port and converts each valid date to a date value. If the value is not a valid date, the Data Integration Service returns ERROR and skips the row.

This example returns a Date/Time value. Therefore, the output port for the expression needs to be Date/Time:

```
IIF( IS_DATE ( INVOICE_DATE, 'YYYY/MM/DD' ), TO_DATE( INVOICE_DATE ), ERROR('Not a valid date' ) )
```

INVOICE_DATE	RETURN VALUE
NULL	NULL
'180'	'Not a valid date'
'04/01/98'	'Not a valid date'
'1998/01/12'	1998/01/12
'1998/11/21 00:00:13'	'Not a valid date'
'1998/02/31'	'Not a valid date'
'John Smith'	'Not a valid date'

IS_NUMBER

Returns whether a string is a valid number. A valid number consists of the following parts:

- Optional space before the number
- Optional sign (+/-)
- One or more digits with an optional decimal point
- Optional scientific notation, such as the letter ‘e’ or ‘E’ (and the letter ‘d’ or ‘D’ on Windows) followed by an optional sign (+/-), followed by one or more digits

- Optional white space following the number

The following numbers are all valid:

```
' 100  '
'   +100'
'-100'
'-3.45e+32'
'+3.45E-32'
'+3.45d+32' (Windows only)
'+3.45D-32' (Windows only)
'.6804'
```

The output port for an IS_NUMBER expression must be a String or Numeric datatype.

You might use IS_NUMBER to test or filter data in a flat file before writing it to a target.

Syntax

```
IS_NUMBER( value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
value	Required	Must be a String datatype. Passes the rows you want to evaluate. You can enter any valid transformation expression.

Return Value

TRUE (1) if the row is a valid number.

FALSE (0) if the row is not a valid number.

NULL if a value in the expression is NULL.

Examples

The following expression checks the ITEM_PRICE port for valid numbers:

```
IS_NUMBER( ITEM_PRICE )
```

ITEM_PRICE	RETURN VALUE
'123.00'	1 (True)
'-3.45e+3'	1 (True)
'-3.45D-3'	1 (True - Windows only)
'-3.45d-3'	0 (False - UNIX only)
'3.45E-'	0 (False) *Incomplete number*
' '	0 (False) *Consists entirely of blanks*
''	0 (False) *Empty string*
'+123abc'	0 (False)
' 123'	1 (True) *Leading white blanks*

ITEM_PRICE	RETURN VALUE
'123 '	1 (True) *Trailing white blanks*
'ABC'	0 (False)
'-ABC'	0 (False)
NULL	NULL

Use IS_NUMBER to test data before using one of the numeric conversion functions, such as TO_FLOAT. For example, the following expression checks the values in the ITEM_PRICE port and converts each valid number to a double-precision floating point value. If the value is not a valid number, the Data Integration Service returns 0.00:

```
IIF( IS_NUMBER ( ITEM_PRICE ), TO_FLOAT( ITEM_PRICE ), 0.00 )
```

ITEM_PRICE	RETURN VALUE
'123.00'	123
'-3.45e+3'	-3450
'3.45E-3'	0.00345
' '	0.00 *Consists entirely of blanks*
''	0.00 *Empty string*
'+123abc'	0.00
'' 123ABC'	0.00
'ABC'	0.00
'-ABC'	0.00
NULL	NULL

IS_SPACES

Returns whether a string value consists entirely of spaces. A space is a blank space, a formfeed, a newline, a carriage return, a tab, or a vertical tab.

IS_SPACES evaluates an empty string as FALSE because there are no spaces. To test for an empty string, use LENGTH.

Syntax

```
IS_SPACES( value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
value	Required	Must be a string datatype. Passes the rows you want to evaluate. You can enter any valid transformation expression.

Return Value

TRUE (1) if the row consists entirely of spaces.

FALSE (0) if the row contains data.

NULL if a value in the expression is NULL.

Example

The following expression checks the ITEM_NAME port for rows that consist entirely of spaces:

```
IS_SPACES( ITEM_NAME )
```

ITEM_NAME	RETURN VALUE
Flashlight	0 (False)
	1 (True)
Regulator system	0 (False)
NULL	NULL
''	0 (FALSE) (*Empty string does not contain spaces.*)

Tip: Use IS_SPACES to avoid writing spaces to a character column in a target table. For example, if you have a transformation that writes customer names to a fixed length CHAR(5) column in a target table, you might want to write '00000' instead of spaces. You would create an expression similar to the following:

```
IIF( IS_SPACES( CUST_NAMES ), '00000', CUST_NAMES )
```

LAG

Returns the value that is an offset number of rows before the current row in an Expression transformation. Use this function to compare values in the current row with values in a previous row when you run a mapping on the Spark engine in the Hadoop environment.

A lag value appears before the current row in a set of data.

When you use LAG in a transformation, you must configure the transformation for windowing. Windowing properties define how the data is partitioned and ordered.

Syntax

```
LAG ( column_name, offset, default )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
column_name	Required	The target column or expression that the function operates on.
offset	Required	Integer data type. The number of rows before the current row to obtain a value from.
default	Optional	The default value to be returned in case the offset is outside the bounds of the partition or table. Default is NULL.

Return Value

The data type of the specified *column_name*.

Default if the return value is outside the bounds of the specified partition.

NULL if *default* is omitted or set to NULL.

Examples

The following expression returns the date that the previous order was placed:

```
LAG ( ORDER_DATE, 1, NULL )
```

ORDER_DATE	ORDER_ID	RETURN VALUE
2017/09/25	1	NULL
2017/09/26	2	2017/09/25
2017/09/27	3	2017/09/26
2017/09/28	4	2017/09/27
2017/09/29	5	2017/09/28
2017/09/30	6	2017/09/29

The lag value of the first row is outside the partition, so the function returned the default value of NULL.

In the following example, your organization receives GPS pings from vehicles that include trip and event IDs and a time stamp. You want to calculate the time difference between each ping.

The following expression calculates the time difference between the current row and the previous row for two separate trips:

```
DATE_DIFF( EVENT_TIME, LAG ( EVENT_TIME, 1, NULL ), 'ss' )
```

You partition the data by trip ID and order by event ID.

TRIP_ID	EVENT_ID	EVENT_TIME	RETURN VALUE
101	1	2017-05-03 12:00:00	NULL
101	2	2017-05-03 12:00:34	34
101	3	2017-05-03 12:02:00	86
102	1	2017-05-03 12:00:00	NULL
102	2	2017-05-03 12:01:56	116
102	3	2017-05-03 12:02:00	4

The lag values of the first and fourth row are outside the specified partition, so the function returned two default NULL values.

LAST

Returns the last row in the selected port. Optionally, you can apply a filter to limit the rows the Data Integration Service reads. You can nest only one other aggregate function within LAST.

Syntax

```
LAST( value [, filter_condition ] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Passes the values for which you want to return the last row. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Last row in a port.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Example

The following expression returns the last row in the ITEMS_NAME port with a price greater than $10.00:

```
LAST( ITEM_NAME, ITEM_PRICE > 10 )
```

ITEM_NAME	ITEM_PRICE
Flashlight	35.00
Navigation Compass	8.05
Regulator System	150.00
Flashlight	29.00
Depth/Pressure Gauge	88.00
Vest	31.00

RETURN VALUE: Vest

LAST_DAY

Returns the date of the last day of the month for each date in a port.

Syntax

```
LAST_DAY( date )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. Passes the dates for which you want to return the last day of the month. You can enter any valid transformation expression that evaluates to a date.

Return Value

Date. The last day of the month for that date value you pass to this function.

NULL if a value in the selected port is NULL.

Null

If a value is NULL, LAST_DAY ignores the row. However, if all values passed from the port are NULL, LAST_DAY returns NULL.

Group By

LAST_DAY groups values based on group by ports you define in the transformation, returning one result for each group. If there is no group by port, LAST_DAY treats all rows as one group, returning one value.

Examples

The following expression returns the last day of the month for each date in the ORDER_DATE port:

```
LAST_DAY( ORDER_DATE )
```

ORDER_DATE	RETURN VALUE
Apr 1 1998 12:00:00AM	Apr 30 1998 12:00:00AM
Jan 6 1998 12:00:00AM	Jan 31 1998 12:00:00AM
Feb 2 1996 12:00:00AM	Feb 29 1996 12:00:00AM *(Leap year)*
NULL	NULL
Jul 31 1998 12:00:00AM	Jul 31 1998 12:00:00AM

You can nest TO_DATE to convert string values to a date. TO_DATE always includes time information. If you pass a string that does not have a time value, the date returned will include the time 00:00:00.

The following example returns the last day of the month for each order date in the same format as the string:

```
LAST_DAY( TO_DATE( ORDER_DATE, 'DD-MON-YY' ))
```

ORDER_DATE	RETURN VALUE
'18-NOV-98'	Nov 30 1998 00:00:00
'28-APR-98'	Apr 30 1998 00:00:00
NULL	NULL
'18-FEB-96'	Feb 29 1996 00:00:00 *(Leap year)*

LEAD

Returns the value that is an offset number of rows after the current row in an Expression transformation. Use this function to compare values in the current row with values in a future row when you run a mapping on the Spark engine in the Hadoop environment.

A lead value appears after the current row in a set of data.

Note: When you use LEAD in a transformation, you must configure the transformation for windowing. Windowing properties define how the data is partitioned and ordered.

Syntax

```
LEAD ( column_name, offset, default )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
column_name	Required	The target column or expression that the function operates on.
offset	Required	Integer data type. The number of rows after the current row to obtain a value from.
default	Optional	The default value to be returned in case the offset is outside the bounds of the partition or table. Default is NULL.

Return Value

The data type of the specified *column_name*.

Default if the return value is outside the bounds of the specified partition.

NULL if *default* is omitted or set to NULL.

Examples

The following expression returns, for each employee, the date the next employee was hired:

```
LEAD ( HIRE_DATE, 1, NULL )
```

EMPLOYEE	HIRE_DATE	RETURN VALUE
Hynes	2012/12/07	2014/05/18
Williams	2014/05/18	2015/07/24
Pritchard	2015/07/24	2015/12/24
Snyder	2015/12/24	2016/11/15
Troy	2016/11/15	2017/08/10
Randolph	2017/08/10	NULL

There is no lead value available for the last row, so the function returned the default value of NULL.

The following expression returns the difference in sales quota values between the first quarter to the third quarter of two calendar years:

```
LEAD ( Sales_Quota, 2, 0 ) - Sales_Quota
```

You partition the data by year and order by quarter.

YEAR	QUARTER	SALES_QUOTA	QUOTA_DIFF
2016	1	300	7700

YEAR	QUARTER	SALES_QUOTA	QUOTA_DIFF
2016	2	7000	0
2016	3	8000	0
2017	1	5000	4000
2017	2	400	0
2017	3	9000	0

The lead values of the second and third quarter are outside the specified partition, so the function returned a value of "0."

LEAST

Returns the smallest value from a list of input values. By default, the match is case sensitive.

Syntax

```
LEAST( value1, [value2, ..., valueN,] CaseFlag )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Any datatype except Binary. Datatype must be compatible with other values. Value you want to compare against other values. You must enter at least one value argument. If the value is Numeric, and other input values are of other numeric datatypes, all values use the highest precision possible. For example, if some values are of the Integer datatype and others are of the Double datatype, the Data Integration Service converts the values to Double.
CaseFlag	Optional	Must be an integer. Specify a value when the input value argument is a string value. Determines whether the arguments in this function are case sensitive. You can enter any valid transformation expression. When CaseFlag is a number other than 0, the function is case sensitive. When CaseFlag is 0, the function is not case sensitive. Default is case sensitive.

Return Value

value1 if it is the smallest of the input values, *value2* if it is the smallest of the input values, and so on.

NULL if any of the arguments is null.

Example

The following expression returns the smallest quantity of items ordered:

```
LEAST( QUANTITY1, QUANTITY2, QUANTITY3 )
```

QUANTITIY1	QUANTITY2	QUANTITY3	RETURN VALUE
150	756	27	27
			NULL
5000	97	17	17
120	1724	965	120

LENGTH

Returns the number of characters in a string, including trailing blanks.

Syntax

```
LENGTH( string )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
string	Required	String datatype. The strings you want to evaluate. You can enter any valid transformation expression.

Return Value

Integer representing the length of the string.

NULL if a value passed to the function is NULL.

Example

The following expression returns the length of each customer name:

```
LENGTH( CUSTOMER_NAME )
```

CUSTOMER_NAME	RETURN VALUE
Bernice Davis	13
NULL	NULL
John Baer	9
Greg Brown	10

Tips for LENGTH

Use LENGTH to test for empty string conditions. If you want to find fields in which customer name is empty, use an expression such as:

```
IIF( LENGTH( CUSTOMER_NAME ) = 0, 'EMPTY STRING' )
```

To test for a null field, use ISNULL. To test for spaces, use IS_SPACES.

LN

Returns the natural logarithm of a numeric value. For example, `LN(3)` returns `1.098612`. You usually use this function to analyze scientific data rather than business data.

This function is the reciprocal of the function EXP.

Syntax

```
LN( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. It must be a positive number, greater than 0. Passes the values for which you want to calculate the natural logarithm. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the natural logarithm for all values in the NUMBERS port:

```
LN( NUMBERS )
```

NUMBERS	RETURN VALUE
10	2.302585092994
125	4.828313737302
0.96	-0.04082199452026
NULL	NULL
-90	*Error. (The Integration Service does not write row.)*
0	*Error. (The Integration Service does not write row.)*

Note: The Data Integration Service displays an error and does not write the row when you pass a negative number or 0. The *numeric_value* must be a positive number greater than 0.

LOG

Returns the logarithm of a numeric value. Most often, you use this function to analyze scientific data rather than business data.

Syntax

```
LOG( base, exponent )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
base	Required	The base of the logarithm. Must be a positive numeric value other than 0 or 1. Any valid transformation expression that evaluates to a positive number other than 0 or 1.
exponent	Required	The exponent of the logarithm. Must be a positive numeric value greater than 0. Any valid transformation expression that evaluates to a positive number greater than 0.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the logarithm for all values in the NUMBERS port:

```
LOG( BASE, EXPONENT )
```

BASE	EXPONENT	RETURN VALUE
15	1	0
.09	10	-0.956244644696599
NULL	18	NULL
35.78	NULL	NULL
-9	18	*Error. (Data Integration Service does not write the row.)*
0	5	*Error. (Data Integration Service does not write the row.)*
10	-2	*Error. (Data Integration Service does not write the row.)*

The Data Integration Service displays an error and does not write the row if you pass a negative number, 0, or 1 as a base value, or if you pass a negative value for the exponent.

LOWER

Converts uppercase string characters to lowercase.

Syntax

```
LOWER( string )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
string	Required	Any string value. The argument passes the string values that you want to return as lowercase. You can enter any valid transformation expression that evaluates to a string.

Return Value

Lowercase character string. If the data contains multibyte characters, the return value depends on the code page and data movement mode of the Integration Service.

NULL if a value in the selected port is NULL.

Example

The following expression returns all first names to lowercase:

```
LOWER( FIRST_NAME )
```

FIRST_NAME	RETURN VALUE
antonia	antonia
NULL	NULL
THOMAS	thomas
PierRe	pierre
BERNICE	bernice

LPAD

Adds a set of blanks or characters to the beginning of a string to set the string to a specified length.

Syntax

```
LPAD( first_string, length [,second_string] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
first_string	Required	Can be a character string. The strings you want to change. You can enter any valid transformation expression.
length	Required	Must be a positive integer literal. This argument specifies the length you want each string to be.
second_string	Optional	Can be any string value. The characters you want to append to the left-side of the *first_string* values. You can enter any valid transformation expression. You can enter a specific string literal. However, enclose the characters you want to add to the beginning of the string within single quotation marks, as in 'abc'. This argument is case sensitive. If you omit the *second_string*, the function pads the beginning of the first string with blanks.

Return Value

String of the specified length.

NULL if a value passed to the function is NULL or if *length* is a negative number.

Examples

The following expression standardizes numbers to six digits by padding them with leading zeros:

```
LPAD( PART_NUM, 6, '0')
```

PART_NUM	RETURN VALUE
702	000702
1	000001
0553	000553
484834	484834

LPAD counts the length from left to right. If the first string is longer than the length, LPAD truncates the string from right to left. For example, LPAD('alphabetical', 5, 'x') returns the string 'alpha'.

If the second string is longer than the total characters needed to return the specified length, LPAD uses a portion of the second string:

```
LPAD( ITEM_NAME, 16, '*..*' )
```

ITEM_NAME	RETURN VALUE
Flashlight	*..**.Flashlight
Compass	*..**..**Compass
Regulator System	Regulator System
Safety Knife	*..*Safety Knife

LTRIM

Removes blanks or characters from the beginning of a string. You can use LTRIM with IIF or DECODE in an Expression or Update Strategy transformation to avoid spaces in a target table.

If you do not specify a *trim_set* parameter in the expression:

- In UNICODE mode, LTRIM removes both single- and double-byte spaces from the beginning of a string.
- In ASCII mode, LTRIM removes only single-byte spaces.

If you use LTRIM to remove characters from a string, LTRIM compares the *trim_set* to each character in the *string* argument, character-by-character, starting with the left side of the string. If the character in the string matches any character in the *trim_set*, LTRIM removes it. LTRIM continues comparing and removing characters until it fails to find a matching character in the *trim_set*. Then it returns the string, which does not include matching characters.

Syntax

```
LTRIM( string [, trim_set] )
```

The following table describes the arguments for this command:

Arguments	Required/ Optional	Description
string	Required	Any string value. Passes the strings you want to modify. You can enter any valid transformation expression. Use operators to perform comparisons or concatenate strings before removing characters from the beginning of a string.
trim_set	Optional	Any string value. Passes the characters you want to remove from the beginning of the first string. You can enter any valid transformation expression. You can also enter a character string. However, you must enclose the characters you want to remove from the beginning of the string within single quotation marks, for example, 'abc'. If you omit the second string, the function removes any blanks from the beginning of the string. LTRIM is case sensitive. For example, if you want to remove the 'A' character from the string 'Alfredo', you would enter 'A', not 'a'.

Return Value

String. The string values with the specified characters in the *trim_set* argument removed.

NULL if a value passed to the function is NULL. If the *trim_set* is NULL, the function returns NULL.

Example

The following expression removes the characters ‘S’ and ‘.’ from the strings in the LAST_NAME port:

```
LTRIM( LAST_NAME, 'S.')
```

LAST_NAME	RETURN VALUE
Nelson	Nelson
Osborne	Osborne
NULL	NULL

LAST_NAME	RETURN VALUE
S. MacDonald	MacDonald
Sawyer	awyer
H. Bender	H. Bender
Steadman	teadman

LTRIM removes ‘S.’ from S. MacDonald and the ‘S’ from both Sawyer and Steadman, but not the period from H. Bender. This is because LTRIM searches, character-by-character, for the set of characters you specify in the *trim_set* argument. If the first character in the string matches the first character in the *trim_set*, LTRIM removes it. Then LTRIM looks at the second character in the string. If it matches the second character in the *trim_set*, LTRIM removes it, and so on. When the first character in the string does not match the corresponding character in the *trim_set*, LTRIM returns the string and evaluates the next row.

In the example of H. Bender, H does not match either character in the *trim_set* argument, so LTRIM returns the string in the LAST_NAME port and moves to the next row.

Tips for LTRIM

Use RTRIM and LTRIM with || or CONCAT to remove leading and trailing blanks after you concatenate two strings.

You can also remove multiple sets of characters by nesting LTRIM. For example, if you want to remove leading blanks and the character 'T' from a column of names, you might create an expression similar to the following:

```
LTRIM( LTRIM( NAMES ), 'T' )
```

MAKE_DATE_TIME

Returns the date and time based on the input values.

Syntax

```
MAKE_DATE_TIME( year, month, day, hour, minute, second, nanosecond )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
year	Required	Numeric datatype. Positive 4-digit integer. If you pass this function a 2-digit year, the Data Integration Service returns “00” as the first two digits of the year.
month	Required	Numeric datatype. Positive integer between 1 and 12 (January=1 and December=12). **Note:** The Spark engine writes null values for rows when the month argument in the MAKE_DATE_TIME function passes an invalid value. In the native environment, the Data Integration Service rejects the row and does not write it to the target.

Argument	Required/ Optional	Description
day	Required	Numeric datatype. Positive integer between 1 and 31 (except for the months that have less than 31 days: February, April, June, September, and November).
hour	Optional	Numeric datatype. Positive integer between 0 and 24 (where 0=12AM, 12=12PM, and 24 =12AM).
minute	Optional	Numeric datatype. Positive integer between 0 and 59.
second	Optional	Numeric datatype. Positive integer between 0 and 59.
nanosecond	Optional	Numeric datatype. Positive integer between 0 and 999,999,999.

Return Value

Date as MM/DD/YYYY HH24:MI:SS. Returns a null value if you do not pass the function a year, month, or day.

Example

The following expression creates a date and time from the input ports:

```
MAKE_DATE_TIME( SALE_YEAR, SALE_MONTH, SALE_DAY, SALE_HOUR, SALE_MIN, SALE_SEC )
```

SALE_YR	SALE_MTH	SALE_DAY	SALE_HR	SALE_MIN	SALE_SEC	RETURN VALUE
2002	10	27	8	36	22	10/27/2002 08:36:22
2000	6	15	15	17		06/15/2000 15:17:00
2003	1	3		22	45	01/03/2003 00:22:45
04	3	30	12	5	10	03/30/0004 12:05:10
99	12	12	5		16	12/12/0099 05:00:16

MAP

Generates a map with elements based on the specified key-value pair.

Syntax

```
MAP(map_key1 as any, map_value1 as any [, map_key2, map_value2]...)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
map_key1	Required	Any primitive data type. An element that you want to add as a key of the map data. You can enter any valid transformation expression.
map_value1	Required	Any primitive or complex data type. An element that you want to add as a value for the key of the map data. You can enter any valid transformation expression.

If you use the MAP function in an output expression for a map port, the data type of the function arguments must match the data type of the map elements specified in the type configuration for the map port. The map_key cannot be null.

Return Value

Map.

The data type of arguments determines the data type of the map elements. For example, if you pass integer arguments as key and struct arguments as value, the function generates map data with a key-value pair of integer and struct types.

Examples

The following expression generates a map of integer and string elements.

```
MAP(emp_id, emp_name)
```

emp_id	emp_name	RETURN VALUE
45781	'Laura'	[45781 -> 'Lauren']
78345	'Derrick'	[78345 -> 'Derrick']
87289	'Kevin'	[87289 -> 'Kevin']
30912		[30912 -> NULL]

MAP_FROM_ARRAYS

Generates a map from the specified key and value arrays.

Syntax

```
MAP_FROM_ARRAYS(map_keys as ARRAY, map_values as ARRAY)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
map_keys	Required	Array type with elements of primitive data type. Array of elements that you want to add as keys of the map data. You can enter any valid transformation expression.
map_values	Required	Array type with elements of any data type. Array of elements that you want to add as values for keys of the map data. You can enter any valid transformation expression.

The number of elements in the map_keys array and map_values array must match. If you use the MAP function in an output expression for a map port, the data type of the function arguments must match the data type of the map elements specified in the type configuration for the map port.

Return Value

Map.

The data type of arguments determines the data type of the map elements. For example, if you pass integer arguments as key and struct arguments as value, the function generates map data with a key-value pair of integer and struct types.

Examples

The following expression generates a map from the array elements of string type.

```
MAP_FROM_ARRAYS(cust_name, cust_phone)
```

```
cust_typecust_name     cust_phone              RETURN VALUE

silver   [Adams, Clark][205-128-6478,          [Adams -> 205-128-6478, Clark ->
                       722-515-2889]           722-515-2889]

gold     [Baker, Davis][107-081-0961,          [Baker -> 107-081-0961, Davis ->
                       718-051-8116]           718-051-8116]

platinum [Evans, Hills][344-894-6463,          [Evans -> 344-894-6463, Hills ->
                       861-411-8361]           861-411-8361]
```

MAP_KEYS

Returns an array of key elements for the specified map.

Syntax

```
MAP_KEYS(map as MAP)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
map	Required	Map data type. Map with key-value pair elements from which you want to retrieve the keys. You can enter any valid transformation expression.

Return Value

Array.

The data type of keys determines the data type of the array elements. For example, if the key is of string type, the function generates array data with string elements.

Returns -1 if the map key is null.

Returns NULL if the map is null.

Examples

The following expression generates an array with elements of string type.

```
MAP_KEYS(stock_sellprice)
```

stock_sellprice	RETURN VALUE
[AAPL -> 150.45, GOOGL -> 1150.96]	[AAPL, GOOGL]
[AMZN -> 1400.54, TSLA -> 339.63]	[AMZN, TSLA]

MAP_VALUES

Returns an array of value elements for the specified map.

Syntax

```
MAP_KEYS(map as MAP)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
map	Required	Map data type. Map with key-value pair elements from which you want to retrieve the values. You can enter any valid transformation expression.

Return Value

Array.

The data type of values in the map determines the data type of the array elements. For example, if the value is of integer type, the function generates array data with integer elements.

Returns -1 if the map value is null.

Returns NULL if the map is null.

Examples

The following expression generates an array with elements of string type.

```
MAP_VALUES(stock_sellprice)
```

stock_sellprice	RETURN VALUE
[AAPL -> 150.45, GOOGL -> 1150.96]	[150.45, 1150.96]
[AMZN -> 1400.54, TSLA -> 339.63]	[1400.54, 339.63]

MAX (Dates)

Returns the latest date found within a port or group. You can apply a filter to limit the rows in the search. You can nest only one other aggregate function within MAX.

You can also use MAX to return the largest numeric value or the highest string value in a port or group.

Syntax

```
MAX( date [, filter_condition] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. Passes the date for which you want to return a maximum date. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Date.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Example

You can return the maximum date for a port or group. The following expression returns the maximum order date for flashlights:

```
MAX( ORDERDATE, ITEM_NAME='Flashlight' )
```

ITEM_NAME	ORDER_DATE
Flashlight	Apr 20 1998
Regulator System	May 15 1998

ITEM_NAME	ORDER_DATE
Flashlight	Sep 21 1998
Diving Hood	Aug 18 1998
Flashlight	NULL

MAX (Numbers)

Returns the maximum numeric value found within a port or group. You can apply a filter to limit the rows in the search. You can nest only one other aggregate function within MAX. You can also use MAX to return the latest date or the highest string value in a port or group.

Syntax

```
MAX( numeric_value [, filter_condition] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the numeric values for which you want to return a maximum numeric value. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Nulls

If a value is NULL, MAX ignores it. However, if all values passed from the port are NULL, MAX returns NULL.

Group By

MAX groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, MAX treats all rows as one group, returning one value.

Example

The first expression returns the maximum price for flashlights:

```
MAX( PRICE, ITEM_NAME='Flashlight' )
```

ITEM_NAME	PRICE
Flashlight	10.00
Regulator System	360.00
Flashlight	55.00
Diving Hood	79.00
Halogen Flashlight	162.00
Flashlight	85.00
Flashlight	NULL

RETURN VALUE: 85.00

MAX (String)

Returns the highest string value found within a port or group. You can apply a filter to limit the rows in the search. You can nest only one other aggregate function within MAX.

Note: The MAX function uses the same sort order that the Sorter transformation uses. However, the MAX function is case sensitive, and the Sorter transformation may not be case sensitive.

You can also use MAX to return the latest date or the largest numeric value in a port or group.

Syntax

```
MAX( string [, filter_condition] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
string	Required	String datatype. Passes the string values for which you want to return a maximum string value. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

String.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Nulls

If a value is NULL, MAX ignores it. However, if all values passed from the port are NULL, MAX returns NULL.

Group By

MAX groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, MAX treats all rows as one group, returning one value.

Example

The following expression returns the maximum item name for manufacturer ID 104:

```
MAX( ITEM_NAME, MANUFACTURER_ID='104' )
```

MANUFACTURER_ID	ITEM_NAME
101	First Stage Regulator
102	Electronic Console
104	Flashlight
104	Battery (9 volt)
104	Rope (20 ft)
104	60.6 cu ft Tank
107	75.4 cu ft Tank
108	Wristband Thermometer

RETURN VALUE: Rope (20 ft)

MD5

Calculates the checksum of the input value. The function uses Message-Digest algorithm 5 (MD5). MD5 is a one-way cryptographic hash function with a 128-bit hash value. You can conclude that input values are different when the checksums of the input values are different. Use MD5 to verify data integrity.

Syntax

```
MD5( value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
value	Required	String or Binary datatype. Value for which you want to calculate checksum. The case of the input value affects the return value. For example, MD5(informatica) and MD5(Informatica) return different values.

Return Value

Unique 32-character string of hexadecimal digits 0-9 and a-f.

NULL if the input is a null value.

Example

You want to write changed data to a database. Use MD5 to generate checksum values for rows of data you read from a source. When you run a mapping, compare the previously generated checksum values against the new checksum values. Then, write the rows with updated checksum values to the target. You can conclude that an updated checksum value indicates that the data has changed.

Tip

You can use the return value as a hash key.

MEDIAN

Returns the median of all values in a selected port.

If there is an even number of values in the port, the median is the average of the middle two values when all values are placed ordinally on a number line. If there is an odd number of values in the port, the median is the middle number.

You can nest only one other aggregate function within MEDIAN, and the nested function must return a Numeric datatype.

The Data Integration Service reads all rows of data to perform the median calculation. The process of reading rows of data to perform the calculation may affect performance. Optionally, you can apply a filter to limit the rows you read to calculate the median.

Syntax

```
MEDIAN( numeric_value [, filter_condition ] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values for which you want to calculate a median. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if no rows are selected. For example, the filter condition evaluates to FALSE or NULL for all rows.

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Nulls

If a value is NULL, MEDIAN ignores the row. However, if all values passed from the port are NULL, MEDIAN returns NULL.

Group By

MEDIAN groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, MEDIAN treats all rows as one group, returning one value.

Example

To calculate the median salary for all departments, you create an Aggregator transformation grouped by departments with a port specifying the following expression:

```
MEDIAN( SALARY )
```

The following expression returns the median value for orders of stabilizing vests:

```
MEDIAN( SALES, ITEM = 'Stabilizing Vest' )
```

ITEM	SALES
Flashlight	85
Stabilizing Vest	504
Stabilizing Vest	36
Safety Knife	5
Medium Titanium Knife	150
Tank	NULL
Stabilizing Vest	441

ITEM	SALES
Chisel Point Knife	60
Stabilizing Vest	NULL
Stabilizing Vest	1044
Wrist Band Thermometer	110

RETURN VALUE: 472.5

METAPHONE

Encodes string values. You can specify the length of the string that you want to encode.

METAPHONE encodes characters of the English language alphabet (A-Z). It encodes both uppercase and lowercase letters in uppercase.

METAPHONE encodes characters according to the following list of rules:

- Skips vowels (A, E, I, O, and U) unless one of them is the first character of the input string. METAPHONE('CAR') returns 'KR' and METAPHONE('AAR') returns 'AR'.
- Uses special encoding guidelines.

The following table lists the METAPHONE encoding guidelines:

Input	Returns	Condition	Example
B	- n/a	- when it follows M	- METAPHONE ('Lamb') returns LM.
B	- B	- in all other cases	- METAPHONE ('Box') returns BKS.
C	- X	- when followed by IA or H	- METAPHONE ('Facial') returns FXL.
C	- S	- when followed by I, E, or Y	- METAPHONE ('Fence') returns FNS.
C	- n/a	- when it follows S, and is followed by I, E, or Y	- METAPHONE ('Scene') returns SN.
C	- K	- in all other cases	- METAPHONE ('Cool') returns KL.
D	- J	- when followed by GE, GY, or GI	- METAPHONE ('Dodge') returns TJ.
D	- T	- in all other cases	- METAPHONE ('David') returns TFT.
F	- F	- in all cases	- METAPHONE ('FOX') returns FKS.
G	- F	- when followed by H and the first character in the input string is not B, D, or H	- METAPHONE ('Tough') returns TF.
G	- n/a	- when followed by H and the first character in the input string is B, D, or H	- METAPHONE ('Hugh') returns HF.

Input	Returns	Condition	Example
G	- J	- when followed by I, E or Y and does not repeat	- METAPHONE (‘Magic’) returns MJK.
G	- K	- in all other cases	- METAPHONE(‘GUN’) returns KN.
H	- H	- when it does not follow C, G, P, S, or T and is followed by A, E, I, or U	- METAPHONE (‘DHAT’) returns THT.
H	- n/a	- in all other cases	- METAPHONE (‘Chain’) returns XN.
J	- J	- in all cases	- METAPHONE (‘Jen’) returns JN.
K	- n/a - K	- when it follows C - in all other cases	- METAPHONE (‘Ckim’) returns KM. - METAPHONE (‘Kim’) returns KM.
L	- L	- in all cases	- METAPHONE (‘Laura’) returns LR.
M	- M	- in all cases	- METAPHONE (‘Maggi’) returns MK.
N	- N	- in all cases	- METAPHONE (‘Nancy’) returns NNS.
P	- F	- when followed by H	- METAPHONE (‘Phone’) returns FN.
P	- P	- in all other cases	- METAPHONE (‘Pip’) returns PP.
Q	- K	- in all cases	- METAPHONE (‘Queen’) returns KN.
R	- R	- in all cases	- METAPHONE (‘Ray’) returns R.
S	- X	- when followed by H, IO, IA, or CHW	- METAPHONE (‘Cash’) returns KX.
S	- S	- in all other cases	- METAPHONE (‘Sing’) returns SNK.
T	- X	- when followed by IA or IO	- METAPHONE (‘Patio’) returns PX.
T	- 0[1]	- when followed by H	- METAPHONE (‘Thor’) returns 0R.
T	- n/a	- when followed by CH	- METAPHONE (‘Glitch’) returns KLTX.
T	- T	- in all other cases	- METAPHINE (‘Tim’) returns TM.
V	- F	- in all cases	- METAPHONE (‘Vin’) returns FN.
W	- W	- when followed by A, E, I, O, or U	- METAPHONE (‘Wang’) returns WNK.
W	- n/a	- in all other cases	- METAPHONE (‘When’) returns HN.
X	- KS	- in all cases	- METAPHONE (‘Six’) returns SKS.
Y	- Y	- when followed by A, E, I, O, or U	- METAPHONE (‘Yang’) returns YNK.

Input	Returns	Condition	Example
Y	- n/a	- in all other cases	- METAPHONE ('Bobby') returns BB.
Z	- S	- in all cases	- METAPHONE ('Zack') returns SK.

1. The integer 0.

- Skips the initial character and encodes the remaining string if the first two characters of the input string have one of the following values:
  - **KN.** For example, METAPHONE('KNOT') returns 'NT'.
  - **GN.** For example, METAPHONE('GNOB') returns 'NB'.
  - **PN.** For example, METAPHONE('PNRX') returns 'NRKS'.
  - **AE.** For example, METAPHONE('AERL') returns 'ERL'.
- If a character other than "C" occurs more than once in the input string, encodes the first occurrence only. For example, METAPHONE('BBOX') returns 'BKS' and METAPHONE('CCOX') returns 'KKKS'.

Syntax

```
METAPHONE( string [,length] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
string	Required	Must be a character string. Passes the value you want to encode. The first character must be a character in the English language alphabet (A-Z). You can enter any valid transformation expression. Skips any non-alphabetic character in *string*.
length	Optional	Must be an integer greater than 0. Specifies the number of characters in *string* that you want to encode. You can enter any valid transformation expression. When *length* is 0 or a value greater than the length of *string*, encodes the entire input string. Default is 0.

Return Value

String.

NULL if one of the following conditions is true:

- All values passed to the function are NULL.
- No character in *string* is a letter of the English alphabet.
- *string* is empty.

Examples

The following expression encodes the first two characters in EMPLOYEE_NAME port to a string:

```
METAPHONE( EMPLOYEE_NAME, 2 )
```

Employee_Name	Return Value
John	JH
*@#$	NULL
P$%%oc&&KMNL	PK

The following expression encodes the first four characters in EMPLOYEE_NAME port to a string:

```
METAPHONE( EMPLOYEE_NAME, 4 )
```

Employee_Name	Return Value
John	JHN
1ABC	ABK
*@#$	NULL
P$%%oc&&KMNL	PKKM

MIN (Dates)

Returns the earliest date found in a port or group. You can apply a filter to limit the rows in the search. You can nest only one other aggregate function within MIN, and the nested function must return a date datatype.

You can also use MIN to return the smallest numeric value or the lowest string value in a port or group.

Syntax

```
MIN( date [, filter_condition] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. Passes the values for which you want to return minimum value. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Date if the *value* argument is a date.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Nulls

If a single value is NULL, MIN ignores it. However, if all values passed from the port are NULL, MIN returns NULL.

Group By

MIN groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, MIN treats all rows as one group, returning one value.

Example

The following expression returns the oldest order date for flashlights:

```
MIN( ORDER_DATE, ITEM_NAME='Flashlight' )
```

ITEM_NAME	ORDER_DATE
Flashlight	Apr 20 1998
Regulator System	May 15 1998
Flashlight	Sep 21 1998
Diving Hood	Aug 18 1998
Halogen Flashlight	Feb 1 1998
Flashlight	Oct 10 1998
Flashlight	NULL

RETURN VALUE: Apr 20 1998

MIN (Numbers)

Returns the smallest numeric value found in a port or group. You can apply a filter to limit the rows in the search. You can nest only one other aggregate function within MIN, and the nested function must return a numeric datatype.

You can also use MIN to return the latest date or the lowest string value in a port or group.

Syntax

```
MIN( numeric_value [, filter_condition] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatypes. Passes the values for which you want to return minimum value. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Nulls

If a single value is NULL, MIN ignores it. However, if all values passed from the port are NULL, MIN returns NULL.

Group By

MIN groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, MIN treats all rows as one group, returning one value.

Example

The following expression returns the minimum price for flashlights:

```
MIN ( PRICE, ITEM_NAME='Flashlight' )
```

ITEM_NAME	PRICE
Flashlight	10.00
Regulator System	360.00
Flashlight	55.00
Diving Hood	79.00
Halogen Flashlight	162.00
Flashlight	85.00
Flashlight	NULL

RETURN VALUE: 10.00

MIN (String)

Returns the lowest string value found in a port or group. You can apply a filter to limit the rows in the search. You can nest only one other aggregate function within MIN, and the nested function must return a string datatype.

Note: The MIN function uses the same sort order that the Sorter transformation uses. However, the MIN function is case sensitive, but the Sorter transformation may not be case sensitive.

You can also use MIN to return the latest date or the minimum numeric value in a port or group.

Syntax

```
MIN( string [, filter_condition] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
string	Required	String datatype. Passes the values for which you want to return minimum value. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

String value.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Nulls

If a single value is NULL, MIN ignores it. However, if all values passed from the port are NULL, MIN returns NULL.

Group By

MIN groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, MIN treats all rows as one group, returning one value.

Example

The following expression returns the minimum item name for manufacturer ID 104:

```
MIN ( ITEM_NAME, MANUFACTURER_ID='104' )
```

MANUFACTURER_ID	ITEM_NAME
101	First Stage Regulator
102	Electronic Console
104	Flashlight

MANUFACTURER_ID	ITEM_NAME
104	Battery (9 volt)
104	Rope (20 ft)
104	60.6 cu ft Tank
107	75.4 cu ft Tank
108	Wristband Thermometer

RETURN VALUE: 60.6 cu ft Tank

MOD

Returns the remainder of a division calculation. For example, `MOD(8,5)` returns 3.

Syntax

```
MOD( numeric_value, divisor )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. The values you want to divide. You can enter any valid transformation expression.
divisor	Required	The numeric value you want to divide by. The divisor cannot be 0.

Return Value

Numeric value of the datatype you pass to the function. The remainder of the numeric value divided by the divisor.

NULL if a value passed to the function is NULL.

Examples

The following expression returns the modulus of the values in the PRICE port divided by the values in the QTY port:

```
MOD( PRICE, QTY )
```

PRICE	QTY	RETURN VALUE
10.00	2	0

PRICE	QTY	RETURN VALUE
12.00	5	2
9.00	2	1
15.00	3	0
NULL	3	NULL
20.00	NULL	NULL
25.00	0	*Error. Integration Service does not write row.*

The last row (25, 0) produced an error because you cannot divide by 0. To avoid dividing by 0, you can create an expression similar to the following, which returns the modulus of Price divided by Quantity only if the quantity is not 0. If the quantity is 0, the function returns NULL:

```
MOD( PRICE, IIF( QTY = 0, NULL, QTY ))
```

PRICE	QTY	RETURN VALUE
10.00	2	0
12.00	5	2
9.00	2	1
15.00	3	0
NULL	3	NULL
20.00	NULL	NULL
25.00	0	NULL

The last row (25, 0) produced a NULL rather than an error because the IIF function replaces NULL with the 0 in the QTY port.

MOVINGAVG

Returns the average (row-by-row) of a specified set of rows. Optionally, you can apply a condition to filter rows before calculating the moving average.

Syntax

```
MOVINGAVG( numeric_value, rowset [, filter_condition] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. The values for which you want to calculate a moving average. You can enter any valid transformation expression.
rowset	Required	Must be a positive integer literal greater than 0. Defines the row set for which you want to calculate the moving average. For example, if you want to calculate a moving average for a column of data, five rows at a time, you might write an expression such as: `MOVINGAVG(SALES, 5)`.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Nulls

MOVINGAVG ignores null values when calculating the moving average. However, if all values are NULL, the function returns NULL.

Example

The following expression returns the average order for a Stabilizing Vest, based on the first five rows in the Sales port, and thereafter, returns the average for the last five rows read:

```
MOVINGAVG( SALES, 5 )
```

ROW_NO	SALES	RETURN VALUE
1	600	NULL
2	504	NULL
3	36	NULL
4	100	NULL
5	550	358
6	39	245.8
7	490	243

The function returns the average for a set of five rows: 358 based on rows 1 through 5, 245.8 based on rows 2 through 6, and 243 based on rows 3 through 7.

MOVINGSUM

Returns the sum (row-by-row) of a specified set of rows.

Optionally, you can apply a condition to filter rows before calculating the moving sum.

Syntax

```
MOVINGSUM( numeric_value, rowset [, filter_condition] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. The values for which you want to calculate a moving sum. You can enter any valid transformation expression.
rowset	Required	Must be a positive integer literal greater than 0. Defines the rowset for which you want to calculate the moving sum. For example, if you want to calculate a moving sum for a column of data, five rows at a time, you might write an expression such as: `MOVINGSUM( SALES, 5 )`
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if the function does not select any rows (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Nulls

MOVINGSUM ignores null values when calculating the moving sum. However, if all values are NULL, the function returns NULL.

Example

The following expression returns the sum of orders for a Stabilizing Vest, based on the first five rows in the Sales port, and thereafter, returns the average for the last five rows read:

```
MOVINGSUM( SALES, 5 )
```

ROW_NO	SALES	RETURN VALUE
1	600	NULL
2	504	NULL
3	36	NULL
4	100	NULL
5	550	1790

ROW_NO	SALES	RETURN VALUE
6	39	1229
7	490	1215

The function returns the sum for a set of five rows: 1790 based on rows 1 through 5, 1229 based on rows 2 through 6, and 1215 based on rows 3 through 7.

NPER

Returns the number of periods for an investment based on a constant interest rate and periodic, constant payments.

Syntax

```
NPER( rate, present value, payment [, future value, type] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
rate	Required	Numeric. Interest rate earned in each period. Expressed as a decimal number. Divide the rate by 100 to express it as a decimal number.
present value	Required	Numeric. Lump-sum amount a series of future payments is worth.
payment	Required	Numeric. Payment amount due per period. Must be a negative number.
future value	Optional	Numeric. Cash balance you want to attain after the last payment is made. If you omit this value, NPER uses 0.
type	Optional	Boolean. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, the Data Integration Service treats the value as 1.

Return Value

Numeric.

Example

The present value of an investment is $500. Each payment is $2000 and the future value of the investment is $20,000. The following expression returns 9 as the number of periods for which you need to make the payments:

```
NPER ( 0.015, -500, -2000, 20000, TRUE )
```

Notes

To calculate interest rate earned in each period, divide the annual rate by the number of payments made in an year. For example, if you make monthly payments at an annual interest rate of 15 percent, the value of the Rate argument is 15% divided by 12. If you make annual payments, the value of the Rate argument is 15%.

The payment value and present value are negative because these are amounts that you pay.

PARSE_JSON

Parses JSON hierarchical data in a string data type and generates a struct. The struct schema is based on the specified complex data type definition that you pass in the argument. This function is appropriate when you receive hierarchical data midstream in the mapping, and need to parse the data for downstream processing.

Syntax

```
PARSE_JSON(upstream_string,:Type.type_definition_library.type_definition)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
upstream_string	Required	The source string field, midstream in the mapping, that contains hierarchical data in JSON format.
:Type.type_definition_library.type_definition	Required	The complex data type definition that represents the schema of the struct data. Use the reference qualifier `:Type` to reference the type definition library that contains the complex data type definition.

Return Value

Struct.

Examples

The following expression generates a struct based on the specified complex data type definition **customer_def**.

```
PARSE_JSON(upstream_string,:Type.type_definition_library.customer_def)
```

The following example shows JSON data in the upstream fields, assigned as a string data type:

```
{"customer" : "ABC Co", "contract" : "1010", "city" : "Phoenix",  "saleamt" :
"150000.00"}
{"customer" : "Data Inc", "contract" : "1111", "city" : "Portland", "saleamt" :
"20000.00"}
```

The complex data type definition **customer_def** is defined in the type definition library as follows:

```
customer_def{
    customer : string
    contract : int
    city : string
    saleamt : int
}
```

The following table shows how the PARSE_JSON function in the mapping expression parses the upstream string using **customer_def** and returns a struct:

customer	contract	city	saleamt	RETURN VALUE

ABC Co	1010	Phoenix	150000.00	{
				customer:ABC Co
				contract:1010
				city:Phoenix
				saleamt:150000.00
				}

Data Inc	1111	Portland	20000.00	{
				customer:Data Inc
				contract:1111
				city:Portland
				saleamt:20000.00
				}

PARSE_XML

Parses XML hierarchical data in a string data type and generates a struct. The struct schema is based on the specified complex data type definition that you pass in the argument. This function is appropriate when you receive hierarchical data midstream in the mapping, and need to parse the data for downstream processing.

Syntax

```
PARSE_XML(upstream_string,:Type.type_definition_library.type_definition)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
upstream_string	Required	The source string field, midstream in the mapping, that contains hierarchical data in XML format.
:Type.type_definition_library.type_definition	Required	The complex data type definition that represents the schema of the struct data. Use the reference qualifier `:Type` to reference the type definition library that contains the complex data type definition.

Return Value

Struct.

Examples

The following expression generates a struct based on the specified complex data type definition **customer_def**.

```
PARSE_XML(upstream_string,:Type.type_definition_library.customer_def)
```

The following example shows XML data in the upstream fields, assigned as a string data type:

```
<Customers>
```

```
    <Customer>
      <CustName>ABC Co</CustName>
      <Contract>1010</Contract>
      <City>Phoenix</City>
      <SaleAmt>150000.00</SaleAmt>
    </Customer>
    <Customer>
      <CustName>Data Inc</CustName>
      <Contract>1111</Contract>
      <City>Portland</City>
      <SaleAmt>20000.00</SaleAmt>
    </Customer>
  </Customers>
```

The complex data type definition **customer_def** is defined in the type definition library as follows:

```
customer_def{
    CustName : string
    Contract : int
    City : string
    SaleAmt : int
}
```

The following table shows how the PARSE_XML function in the mapping expression parses the upstream string using **customer_def** and returns a struct:

CustName	Contract	City	SaleAmt	RETURN VALUE
ABC Co	1010	Phoenix	150000.00	{ CustName:ABC Co Contract:1010 City:Phoenix SaleAmt:150000.00 }
Data Inc	1111	Portland	20000.00	{ CustName:Data Inc Contract:1111 City:Portland SaleAmt:20000.00 }

PERCENTILE

Calculates the value that falls at a given percentile in a group of numbers. You can nest only one other aggregate function within PERCENTILE, and the nested function must return a Numeric datatype.

The Data Integration Service reads all rows of data to perform the percentile calculation. The process of reading rows to perform the calculation may affect performance. Optionally, you can apply a filter to limit the rows you read to calculate the percentile.

Syntax

```
PERCENTILE( numeric_value, percentile [, filter_condition ] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values for which you want to calculate a percentile. You can enter any valid transformation expression.
percentile	Required	Integer between 0 and 100, inclusive. Passes the percentile you want to calculate. You can enter any valid transformation expression. If you pass a number outside the 0 to 100 range, the Data Integration Service displays an error and does not write the row.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL, or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Nulls

If a value is NULL, PERCENTILE ignores the row. However, if all values in a group are NULL, PERCENTILE returns NULL.

Group By

PERCENTILE groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, PERCENTILE treats all rows as one group, returning one value.

Example

The Data Integration Service calculates a percentile using the following logic:

$$i = \frac{(x + 1) \times \text{percentile}}{100}$$

Use the following guidelines for this equation:

- *x* is the number of elements in the group of values for which you are calculating a percentile.
- If *i* < 1, PERCENTILE returns the value of the first element in the list.
- If *i* is an integer value, PERCENTILE returns the value of the *i*th element in the list.
- Otherwise PERCENTILE returns the value of *n*:

$$n = [\lfloor i \rfloor \text{th?element} \times (\lceil i \rceil - i)] + [\lceil i \rceil \text{th?element} \times (i - \lfloor i \rfloor)]$$

The following expression returns the salary that falls at the 75th percentile of salaries greater than $50,000:

```
PERCENTILE( SALARY, 75, SALARY > 50000 )
```

SALARY
125000.0
27900.0
100000.0
NULL
55000.0
9000.0
85000.0
86000.0
48000.0
99000.0

RETURN VALUE: 106250.0

PMT

Returns the payment for a loan based on constant payments and a constant interest rate.

Syntax

```
PMT( rate, terms, present value[, future value, type] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
rate	Required	Numeric. Interest rate of the loan for each period. Expressed as a decimal number. Divide the rate by 100 to express it as a decimal number.
terms	Required	Numeric. Number of periods or payments. Must be greater than 0. **Note:** The Spark engine writes null values for rows when the terms argument passes a 0 value. In the native environment, the Data Integration Service rejects the row and does not write it to the target.
present value	Required	Numeric. Principle for the loan.

Argument	Required/ Optional	Description
future value	Optional	Numeric. Cash balance you want to attain after the last payment. If you omit this value, PMT uses 0.
type	Optional	Boolean. Timing of the payment. Enter 1 if the payment is at the beginning of period. Enter 0 if the payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, the Data Integration Service treats the value as 1.

Return Value

Numeric.

Example

The following expression returns -2111.64 as the monthly payment amount of a loan:

```
PMT( 0.01, 10, 20000 )
```

Notes

To calculate interest rate earned in each period, divide the annual rate by the number of payments made in a year. For example, if you make monthly payments at an annual interest rate of 15%, the rate is 15%/12. If you make annual payments, the rate is 15%.

The payment value is negative because these are amounts that you pay.

POWER

Returns a value raised to the exponent you pass to the function.

Syntax

```
POWER( base, exponent )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
base	Required	Numeric value. This argument is the base value. You can enter any valid transformation expression. If the base value is negative, the exponent must be an integer.
exponent	Required	Numeric value. This argument is the exponent value. You can enter any valid transformation expression. If the base value is negative, the exponent must be an integer. In this case, the function rounds any decimal values to the nearest integer before returning a value.

Return Value

Double value.

NULL if you pass a null value to the function.

Example

The following expression returns the values in the Numbers port raised to the values in the Exponent port:

```
POWER( NUMBERS, EXPONENT )
```

NUMBERS	EXPONENT	RETURN VALUE
10.0	2.0	100
3.5	6.0	1838.265625
3.5	5.5	982.594307804838
NULL	2.0	NULL
10.0	NULL	NULL
-3.0	-6.0	0.00137174211248285
3.0	-6.0	0.00137174211248285
-3.0	6.0	729.0
-3.0	5.5	729.0

The value -3.0 raised to 6 returns the same results as -3.0 raised to 5.5. If the base is negative, the exponent must be an integer. Otherwise, the Data Integration Service rounds the exponent to the nearest integer value.

PV

Returns the present value of an investment.

Syntax

```
PV( rate, terms, payment [, future value, type] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
rate	Required	Numeric. Interest rate earned in each period. Expresses as a decimal number. Divide the rate by 100 to express it as a decimal number.
terms	Required	Numeric. Number of period or payments. Must be greater than 0. **Note:** The Spark engine writes null values for rows when the terms argument passes a 0 value. In the native environment, the Data Integration Service rejects the row and does not write it to the target.
payments	Required	Numeric. Payment amount due per period. Must be a negative number.

Argument	Required/ Optional	Description
future value	Optional	Numeric. Cash balance after the last payment. If you omit this value, PV uses 0.
types	Optional	Boolean. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if the payment is at the end of period. Default is 0. If you enter a value other than 0 or 1, the Data Integration Service treats the value as 1.

Return Value

Numeric.

Example

The following expression returns 12,524.43 as the amount you must deposit in the account today to have a future value of $20,000 in one year if you also deposit $500 at the beginning of each period:

```
PV( 0.0075, 12, -500, 20000, TRUE )
```

RAND

Returns a random number between 0 and 1. This is useful for probability scenarios.

Syntax

```
RAND( seed )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
seed	Optional	Numeric. Starting value for the Integration Service to generate the random number. Value must be a constant. If you do not enter a seed, the Data Integration Service uses the current system time to derive the numbers of seconds since January 1, 1971. It uses this value as the seed.

Return Value

Numeric.

For the same seed, the Data Integration Service generates the same sequence of numbers.

Example

The following expression may return a value of 0.417022004702574:

```
RAND (1)
```

RATE

Returns the interest rate earned per period by a security.

Syntax

```
RATE( terms, payment, present value[, future value, type] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
terms	Required	Numeric. Number of periods or payments. Must be greater than 0. **Note:** The Spark engine writes null values for rows when the terms argument passes a 0 value. In the native environment, the Data Integration Service rejects the row and does not write it to the target.
payments	Required	Numeric. Payment amount due per period. Must be a negative number.
present value	Required	Numeric. Lump-sum amount that a series of future payments is worth now.
future value	Optional	Numeric. Cash balance you want to attain after the last payment. For example, the future value of a loan is 0. If you omit this argument, RATE uses 0.
types	Optional	Boolean. Timing of the payment. Enter 1 if payment is at the beginning of period. Enter 0 if payment is at the end of the period. Default is 0. If you enter a value other than 0 or 1, the Data Integration Service treats the value as 1.

Return Value

Numeric.

Example

The following expression returns 0.0077 as the monthly interest rate of a loan:

```
RATE( 48, -500, 20000 )
```

To calculate the annual interest rate of the loan, multiply 0.0077 by 12. The annual interest rate is 0.0924 or 9.24%.

REG_EXTRACT

Extracts subpatterns of a regular expression within an input value. For example, from a regular expression pattern for a full name, you can extract the first name or last name.

Note: Use the REG_REPLACE function to replace a character pattern in a string with another character pattern.

Syntax

```
REG_EXTRACT( subject, 'pattern', subPatternNum )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
subject	Required	String datatype. Passes the value you want to compare against the regular expression pattern.
pattern	Required	String datatype. Regular expression pattern that you want to match. You must use perl compatible regular expression syntax. Enclose the pattern in single quotation marks. Enclose each subpattern in parentheses.
subPatternNum	Optional	Integer value. Subpattern number of the regular expression you want to match. Use the following guidelines to determine the subpattern number: - no value or 1. Extracts the first regular expression subpattern. - 2. Extracts the second regular expression subpattern. - n. Extracts the *n*th regular expression subpattern. Default is 1.

Using perl Compatible Regular Expression Syntax

You must use perl compatible regular expression syntax with REG_EXTRACT, REG_MATCH and REG_REPLACE functions.

The following table provides perl compatible regular expression syntax guidelines:

Syntax	Description
. (a period)	Matches any one character.
[a-z]	Matches one instance of a character in lower case. For example, [a-z] matches ab. Use [A-Z] to match characters in upper case.
\d	Matches one instance of any digit from 0-9.
\s	Matches a whitespace character.
\w	Matches one alphanumeric character, including underscore (_)
()	Groups an expression. For example, the parentheses in (\d-\d-\d\d) groups the expression \d\d-\d\d, which finds any two numbers followed by a hyphen and any two numbers, as in 12-34.
{}	Matches the number of characters. For example, \d{3} matches any three numbers, such as 650 or 510. Or, [a-z]{2} matches any two letters, such as CA or NY.
?	Matches the preceding character or group of characters zero or one time. For example, \d{3}(-{d{4})? matches any three numbers, which can be followed by a hyphen and any four numbers.
* (an asterisk)	Matches zero or more instances of the values that follow the asterisk. For example, *0 is any value that precedes a 0.
+	Matches one or more instances of the values that follow the plus sign. For example, \w+ is any value that follows an alphanumeric character.

For example, the following regular expression finds 5-digit U.S.A. zip codes, such as 93930, and 9-digit zip codes, such as 93930-5407:

```
\d{5}(-\d{4})?
```

\d{5} refers to any five numbers, such as 93930. The parentheses surrounding -\d{4} group this segment of the expression. The hyphen represents the hyphen of a 9-digit zip code, as in 93930-5407. \d{4} refers to any four numbers, such as 5407. The question mark states that the hyphen and last four digits are optional or can appear one time.

Converting COBOL Syntax to perl Compatible Regular Expression Syntax

If you are familiar with COBOL syntax, you can use the following information to write perl compatible regular expressions.

The following table shows examples of COBOL syntax and their perl equivalents:

COBOL Syntax	perl Syntax	Description
9	\d	Matches one instance of any digit from 0-9.
9999	\d\d\d\d or \d{4}	Matches any four digits from 0-9, as in 1234 or 5936.
x	[a-z]	Matches one instance of a letter.
9xx9	\d[a-z][a-z]\d	Matches any number followed by two letters and another number, as in 1ab2.

Converting SQL Syntax to perl Compatible Regular Expression Syntax

If you are familiar with SQL syntax, you can use the following information to write perl compatible regular expressions.

The following table shows examples of SQL syntax and their perl equivalents:

SQL Syntax	perl Syntax	Description
%	. *	Matches any string.
A%	A.*	Matches the letter “A” followed by any string, as in Area.
_	. (a period)	Matches any one character.
A_	A.	Matches “A” followed by any one character, such as AZ.

Return Value

Returns the value of the *n*th subpattern that is part of the input value. The *nth* subpattern is based on the value you specify for subPatternNum.

NULL if the input is a null value or if the pattern is null.

Example

You might use REG_EXTRACT in an expression to extract middle names from a regular expression that matches first name, middle name, and last name. For example, the following expression returns the middle name of a regular expression:

```
REG_EXTRACT( Employee_Name, '(\w+)\s+(\w+)\s+(\w+)',2)
```

Employee_Name	Return Value
Stephen Graham Smith	Graham
Juan Carlos Fernando	Carlos

REG_MATCH

Returns whether a value matches a regular expression pattern. This lets you validate data patterns, such as IDs, telephone numbers, postal codes, and state names.

Note: Use the REG_REPLACE function to replace a character pattern in a string with a new character pattern.

Syntax

```
REG_MATCH( subject, pattern )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
subject	Required	String datatype. Passes the value you want to match against the regular expression pattern.
pattern	Required	String datatype. Regular expression pattern that you want to match. You must use perl compatible regular expression syntax. Enclose the pattern in single quotes. For more information, see "REG_EXTRACT" on page 160.

Return Value

TRUE if the data matches the pattern.

FALSE if the data does not match the pattern.

NULL if the input is a null value or if the pattern is NULL.

Example

You might use REG_MATCH in an expression to validate telephone numbers. For example, the following expression matches a 10-digit telephone number against the pattern and returns a Boolean value based on the match:

```
REG_MATCH (Phone_Number, '(\d\d\d-\d\d\d-\d\d\d\d)' )
```

Phone_Number	Return Value
408-555-1212	TRUE
	NULL
510-555-1212	TRUE
92 555 51212	FALSE
650-555-1212	TRUE
415-555-1212	TRUE
831 555 12123	FALSE

Tip

You can also use REG_MATCH for the following tasks:

- To verify that a value matches a pattern. This use is similar to the SQL LIKE function.
- To verify that values are characters. This use is similar to the SQL IS_CHAR function.

To verify that a value matches a pattern, use a period (.) and an asterisk (*) with the REG_MATCH function in an expression. A period matches any one character. An asterisk matches 0 or more instances of values that follow it.

For example, use the following expression to find account numbers that begin with 1835:

```
REG_MATCH(ACCOUNT_NUMBER, '1835.*')
```

To verify that values are characters, use a REG_MATCH function with the regular expression [a-zA-Z]+. a-z matches all lowercase characters. A-Z matches all uppercase characters. The plus sign (+) indicates that there should be at least one character.

For example, use the following expression to verify that a list of last names contain only characters:

```
REG_MATCH(LAST_NAME, '[a-zA-Z]+')
```

REG_REPLACE

Replaces characters in a string with another character pattern. By default, REG_REPLACE searches the input string for the character pattern you specify and replaces all occurrences with the replacement pattern. You can also indicate the number of occurrences of the pattern you want to replace in the string.

Syntax

```
REG_REPLACE( subject, pattern, replace, numReplacements )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
subject	Required	String datatype. Passes the string you want to search.
pattern	Required	String datatype. Passes the character string to be replaced. You must use perl compatible regular expression syntax. Enclose the pattern in single quotes. For more information, see "REG_EXTRACT" on page 160.
replace	Required	String datatype. Passes the new character string.
numReplacements	Optional	Numeric datatype. Specifies the number of occurrences you want to replace. If you omit this option, REG_REPLACE will replace all occurrences of the character string.

Return Value

String

Example

The following expression removes additional spaces from the Employee name data for each row of the Employee_name port:

```
REG_REPLACE( Employee_Name, '\s+', ' ')
```

Employee_Name	RETURN VALUE
Adam Smith	Adam Smith
Greg Sanders	Greg Sanders
Sarah Fe	Sarah Fe
Sam Cooper	Sam Cooper

REPLACECHR

Replaces characters in a string with a single character or no character. REPLACECHR searches the input string for the characters you specify and replaces all occurrences of all characters with the new character you specify.

Syntax

```
REPLACECHR( CaseFlag, InputString, OldCharSet, NewChar )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
CaseFlag	Required	Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid transformation expression. When *CaseFlag* is a number other than 0, the function is case sensitive. When *CaseFlag* is a null value or 0, the function is not case sensitive.
InputString	Required	Must be a character string. Passes the string you want to search. You can enter any valid transformation expression. If you pass a numeric value, the function converts it to a character string. If *InputString* is NULL, REPLACECHR returns NULL.
OldCharSet	Required	Must be a character string. The characters you want to replace. You can enter one or more characters. You can enter any valid transformation expression. You can also enter a text literal enclosed within single quotation marks, for example, 'abc'. If you pass a numeric value, the function converts it to a character string. If *OldCharSet* is NULL or empty, REPLACECHR returns *InputString*.
NewChar	Required	Must be a character string. You can enter one character, an empty string, or NULL. You can enter any valid transformation expression. If *NewChar* is NULL or empty, REPLACECHR removes all occurrences of all characters in *OldCharSet* in *InputString*. If *NewChar* contains more than one character, REPLACECHR uses the first character to replace *OldCharSet*.

Return Value

String.

Empty string if REPLACECHR removes all characters in *InputString*.

NULL if *InputString* is NULL.

InputString if *OldCharSet* is NULL or empty.

Examples

The following expression removes the double quotes from web log data for each row in the WEBLOG port:

```
REPLACECHR( 0, WEBLOG, '"', NULL )
```

WEBLOG	RETURN VALUE
"GET /news/index.html HTTP/1.1"	GET /news/index.html HTTP/1.1
"GET /companyinfo/index.html HTTP/1.1"	GET /companyinfo/index.html HTTP/1.1
GET /companyinfo/index.html HTTP/1.1	GET /companyinfo/index.html HTTP/1.1
NULL	NULL

The following expression removes multiple characters for each row in the WEBLOG port:

```
REPLACECHR ( 1, WEBLOG, '][', NULL )
```

WEBLOG	RETURN VALUE
[29/Oct/2001:14:13:50 -0700]	29/Oct/2001:14:13:50 -0700
[31/Oct/2000:19:45:46 -0700] "GET /news/index.html HTTP/1.1"	31/Oct/2000:19:45:46 -0700 GET /news/index.html HTTP/1.1
[01/Nov/2000:10:51:31 -0700] "GET /news/index.html HTTP/1.1"	01/Nov/2000:10:51:31 -0700 GET /news/index.html HTTP/1.1
NULL	NULL

The following expression changes part of the value of the customer code for each row in the CUSTOMER_CODE port:

```
REPLACECHR ( 1, CUSTOMER_CODE, 'A', 'M' )
```

CUSTOMER_CODE	RETURN VALUE
ABA	MBM
abA	abM
BBC	BBC
ACC	MCC
NULL	NULL

The following expression changes part of the value of the customer code for each row in the CUSTOMER_CODE port:

```
REPLACECHR ( 0, CUSTOMER_CODE, 'A', 'M' )
```

CUSTOMER_CODE	RETURN VALUE
ABA	MBM
abA	MbM
BBC	BBC
ACC	MCC

The following expression changes part of the value of the customer code for each row in the CUSTOMER_CODE port:

```
REPLACECHR ( 1, CUSTOMER_CODE, 'A', NULL )
```

CUSTOMER_CODE	RETURN VALUE
ABA	B

CUSTOMER_CODE	RETURN VALUE
BBC	BBC
ACC	CC
AAA	*[empty string]*
aaa	aaa
NULL	NULL

The following expression removes multiple numbers for each row in the INPUT port:

```
REPLACECHR ( 1, INPUT, '14', NULL )
```

INPUT	RETURN VALUE
12345	235
4141	NULL
111115	5
NULL	NULL

When you want to use a single quote (') in either *OldCharSet* or *NewChar*, you must use the CHR function. The single quote is the only character that cannot be used inside a string literal.

The following expression removes multiple characters, including the single quote, for each row in the INPUT port:

```
REPLACECHR (1, INPUT, CHR(39), NULL )
```

INPUT	RETURN VALUE
'Tom Smith' 'Laura Jones'	Tom Smith Laura Jones
Tom's	Toms
NULL	NULL

REPLACESTR

Replaces characters in a string with a single character, multiple characters, or no character. REPLACESTR searches the input string for all strings you specify and replaces them with the new string you specify.

Syntax

```
REPLACESTR ( CaseFlag, InputString, OldString1, [OldString2, ... OldStringN,] NewString )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
CaseFlag	Required	Must be an integer. Determines whether the arguments in this function are case sensitive. You can enter any valid transformation expression. When *CaseFlag* is a number other than 0, the function is case sensitive. When *CaseFlag* is a null value or 0, the function is not case sensitive.
InputString	Required	Must be a character string. Passes the strings you want to search. You can enter any valid transformation expression. If you pass a numeric value, the function converts it to a character string. If *InputString* is NULL, REPLACESTR returns NULL.
OldString	Required	Must be a character string. The string you want to replace. You must enter at least one *OldString* argument. You can enter one or more characters per *OldString* argument. You can enter any valid transformation expression. You can also enter a text literal enclosed within single quotation marks, for example, 'abc'. If you pass a numeric value, the function converts it to a character string. When REPLACESTR contains multiple *OldString* arguments, and one or more *OldString* arguments is NULL or empty, REPLACESTR ignores the *OldString* argument. When all *OldString* arguments are NULL or empty, REPLACESTR returns *InputString*. The function replaces the characters in the *OldString* arguments in the order they appear in the function. For example, if you enter multiple *OldString* arguments, the first *OldString* argument has precedence over the second *OldString* argument, and the second *OldString* argument has precedence over the third *OldString* argument. When REPLACESTR replaces a string, it places the cursor after the replaced characters in *InputString* before searching for the next match.
NewString	Required	Must be a character string. You can enter one character, multiple characters, an empty string, or NULL. You can enter any valid transformation expression. If *NewString* is NULL or empty, REPLACESTR removes all occurrences of *OldString* in *InputString*.

Return Value

String.

Empty string if REPLACESTR removes all characters in *InputString*.

NULL if *InputString* is NULL.

InputString if all *OldString* arguments are NULL or empty.

Examples

The following expression removes the double quotes and two different text strings from web log data for each row in the WEBLOG port:

```
REPLACESTR( 1, WEBLOG, '"', 'GET ', ' HTTP/1.1', NULL )
```

WEBLOG	RETURN VALUE
"GET /news/index.html HTTP/1.1"	/news/index.html
"GET /companyinfo/index.html HTTP/1.1"	/companyinfo/index.html

WEBLOG	RETURN VALUE
GET /companyinfo/index.html	/companyinfo/index.html
GET	*[empty string]*
NULL	NULL

The following expression changes the title for certain values for each row in the TITLE port:

```
REPLACESTR ( 1, TITLE, 'rs.', 'iss', 's.' )
```

TITLE	RETURN VALUE
Mrs.	Ms.
Miss	Ms.
Mr.	Mr.
MRS.	MRS.

The following expression changes the title for certain values for each row in the TITLE port:

```
REPLACESTR ( 0, TITLE, 'rs.', 'iss', 's.' )
```

TITLE	RETURN VALUE
Mrs.	Ms.
MRS.	Ms.

The following expression shows how the REPLACESTR function replaces multiple `OldString` arguments for each row in the INPUT port:

```
REPLACESTR ( 1, INPUT, 'ab', 'bc', '*' )
```

INPUT	RETURN VALUE
abc	*c
abbc	**
abbbbc	*bb*
bc	*

The following expression shows how the REPLACESTR function replaces multiple `OldString` arguments for each row in the INPUT port:

```
REPLACESTR ( 1, INPUT, 'ab', 'bc', 'b' )
```

INPUT	RETURN VALUE
ab	b

INPUT	RETURN VALUE
bc	b
abc	bc
abbc	bb
abbcc	bbc

When you want to use a single quote (') in either *OldString* or *NewString*, you must use the CHR function. Use both the CHR and CONCAT functions to concatenate a single quote onto a string. The single quote is the only character that cannot be used inside a string literal. Consider the following example:

```
CONCAT( 'Joan', CONCAT( CHR(39), 's car' ))
```

The return value is:

```
Joan's car
```

The following expression changes a string that includes the single quote, for each row in the INPUT port:

```
REPLACESTR ( 1, INPUT, CONCAT('it', CONCAT(CHR(39), 's' )), 'its' )
```

INPUT	RETURN VALUE
it's	its
mit's	mits
mits	mits
mits'	mits'

RESPEC

Renames each element of the given struct value based on the names of the elements in the specified complex data type definition.

Syntax

```
RESPEC(:Type.type_definition_library.type_definition, struct_value)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
:Type.type_definition_library.type_definition	Required	The complex data type definition that represents the schema of the struct data. Use the reference qualifier `:Type` to reference the type definition library that contains the complex data type definition.
struct_value	Required	The struct value for which you want to change the element names. You can enter any valid transformation expression that evaluates to a struct.

The data type of each element in the complex data type definition must match the data type of the corresponding element of the struct.

Return Value

Struct.

Examples

The following expression changes the names of the elements in the struct port h2_sales based on the names in the complex data type definition h1_sales_def.

```
RESPEC(:Type.type_definition_library.h2_sales_def, h2_sales)
```

h2_sales_def	h2_sales	RETURN VALUE

{
q1_sales : int
q2_sales : bigint
}
	{
	q3_total : int
	q4_total : bigint
	}
		{
		q1_sales : int
		q2_sales : bigint
		}

REVERSE

Reverses the input string.

Syntax

```
REVERSE( string )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
string	Required	Any character value. Value you want to reverse.

Return Value

String. Reverse of the input value.

Example

The following expression reverses the numbers of the customer code:

```
REVERSE( CUSTOMER_CODE )
```

CUSTOMER_CODE	RETURN VALUE
0001	1000
0002	2000
0003	3000
0004	4000

ROUND (Dates)

Rounds one part of a date. You can also use ROUND to round numbers.

This function can round the following parts of a date:

Year

Rounds the year portion of a date based on the month.

Month

Rounds the month portion of a date based on the day of the month.

Day

Rounds the day portion of the date based on the time.

Hour

Rounds the hour portion of the date based on the minutes in the hour.

Minute

Rounds the minute portion of the date based on the seconds.

Second

Rounds the second portion of the date based on the milliseconds.

Millisecond

Rounds the millisecond portion of the date based on the microseconds.

Microsecond

Rounds the microsecond portion of the date based on the nanoseconds.

The following table shows the conditions used by the ROUND expression and the return values:

Condition	Expression	Return Value
Month between January and June, function returns January 1 of the same year and sets the time to 00:00:00.000000000.	`ROUND(TO_DATE('04/16/1998 8:24:19','MM/DD/YYYY HH24:MI:SS'),'YY')`	01/01/1998 00:00:00.000000000
Month between July and December, function returns January 1 of next year and sets the time to 00:00:00.000000000.	`ROUND(TO_DATE('07/30/1998 2:30:55','MM/DD/YYYY HH24:MI:SS'),'YY')`	01/01/1999 00:00:00.000000000
Day of the month between 1 and 15, function returns the first day of the input month and sets the time to 00:00:00.000000000.	`ROUND(TO_DATE('04/15/1998 8:24:19','MM/DD/YYYY HH24:MI:SS'),'MM')`	04/01/1998 00:00:00.000000000
Day of the month between 16 and the last day of the month, function returns the first day of the next month and sets the time to 00:00:00.000000000.	`ROUND(TO_DATE('05/22/1998 10:15:29','MM/DD/YYYY HH24:MI:SS'),'MM')`	06/01/1998 00:00:00.000000000
Time between 00:00:00 (12 a.m.) and 11:59:59 a.m., function returns the current date and sets the time to 00:00:00.000000000 (12 a.m.).	`ROUND(TO_DATE('06/13/1998 2:30:45','MM/DD/YYYY HH24:MI:SS'),'DD')`	06/13/1998 00:00:00.000000000
Time 12:00:00 (12 p.m.) or later, function rounds the date to the next day and sets the time to 00:00:00.000000000 (12 a.m.).	`ROUND(TO_DATE('06/13/1998 22:30:45','MM/DD/YYYY HH24:MI:SS'),'DD')`	06/14/1998 00:00:00.000000000
Minute portion of time between 0 and 29 minutes, function returns the current hour and sets minutes, seconds, milliseconds, and nanoseconds to 0.	`ROUND(TO_DATE('04/01/1998 11:29:35','MM/DD/YYYY HH24:MI:SS'),'HH')`	04/01/1998 11:00:00.000000000
Minute portion of the time 30 or greater, function returns the next hour and sets minutes, seconds, milliseconds, and nanoseconds to 0.	`ROUND(TO_DATE('04/01/1998 13:39:00','MM/DD/YYYY HH24:MI:SS'),'HH')`	04/01/1998 14:00:00.000000000
Time between 0 and 29 seconds, function returns the current minute and sets seconds, milliseconds, and nanoseconds to 0.	`ROUND(TO_DATE('05/22/1998 10:15:29','MM/DD/YYYY HH24:MI:SS'),'MI')`	05/22/1998 10:15:00.000000000
Time between 30 and 59 seconds, function returns the next minute and sets seconds, milliseconds, and nanoseconds to 0.	`ROUND(TO_DATE('05/22/1998 10:15:30','MM/DD/YYYY HH24:MI:SS'),'MI')`	05/22/1998 10:16:00.000000000
Time between 0 and 499 milliseconds, function returns the current second and sets milliseconds to 0.	`ROUND(TO_DATE('05/22/1998 10:15:29.499','MM/DD/YYYY HH24:MI:SS.MS'),'SS')`	05/22/1998 10:15:29.000000000
Time between 500 and 999 milliseconds, function returns the next second and sets milliseconds to 0.	`ROUND(TO_DATE('05/22/1998 10:15:29.500','MM/DD/YYYY HH24:MI:SS.MS'),'SS')`	05/22/1998 10:15:30.000000000
Time between 0 and 499 microseconds, function returns the current millisecond and sets microseconds to 0.	`ROUND(TO_DATE('05/22/1998 10:15:29.498125','MM/DD/YYYY HH24:MI:SS.US'),'MS')`	05/22/1998 10:15:29.498000000

Condition	Expression	Return Value
Time between 500 and 999 microseconds, function returns the next millisecond and sets microseconds to 0.	`ROUND(TO_DATE('05/22/1998 10:15:29.498785','MM/DD/YYYY HH24:MI:SS.US'),'MS')`	05/22/1998 10:15:29.499000000
Time between 0 and 499 nanoseconds, function returns the current microsecond and sets nanoseconds to 0.	`ROUND(TO_DATE('05/22/1998 10:15:29.498125345','MM/DD/YYYY HH24:MI:SS.NS'),'US')`	05/22/1998 10:15:29.498125000
Time between 500 and 999 nanoseconds, function returns the next microsecond and sets nanoseconds to 0.	`ROUND(TO_DATE('05/22/1998 10:15:29.498125876','MM/DD/YYYY HH24:MI:SS.NS'),'US')`	05/22/1998 10:15:29.498126000

Syntax

```
ROUND( date [,format] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. You can nest TO_DATE to convert strings to dates before rounding.
format	Optional	Enter a valid format string. This is the portion of the date that you want to round. You can round only one portion of the date. If you omit the format string, the function rounds the date to the nearest day.

Return Value

Date with the specified part rounded. ROUND returns a date in the same format as the source date. You can link the results of this function to any port with a Date/Time datatype.

NULL if you pass a null value to the function.

Examples

The following expressions round the year portion of dates in the DATE_SHIPPED port:

```
ROUND( DATE_SHIPPED, 'Y' )
ROUND( DATE_SHIPPED, 'YY' )
ROUND( DATE_SHIPPED, 'YYY' )
ROUND( DATE_SHIPPED, 'YYYY' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 1 1998 12:00:00.000000000AM
Apr 19 1998 1:31:20PM	Jan 1 1998 12:00:00.000000000AM
Dec 20 1998 3:29:55PM	Jan 1 1999 12:00:00.000000000AM
NULL	NULL

The following expressions round the month portion of each date in the DATE_SHIPPED port:

```
ROUND( DATE_SHIPPED, 'MM' )
ROUND( DATE_SHIPPED, 'MON' )
ROUND( DATE_SHIPPED, 'MONTH' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 1 1998 12:00:00.000000000AM
Apr 19 1998 1:31:20PM	May 1 1998 12:00:00.000000000AM
Dec 20 1998 3:29:55PM	Jan 1 1999 12:00:00.000000000AM
NULL	NULL

The following expressions round the day portion of each date in the DATE_SHIPPED port:

```
ROUND( DATE_SHIPPED, 'D' )
ROUND( DATE_SHIPPED, 'DD' )
ROUND( DATE_SHIPPED, 'DDD' )
ROUND( DATE_SHIPPED, 'DY' )
ROUND( DATE_SHIPPED, 'DAY' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 15 1998 12:00:00.000000000AM
Apr 19 1998 1:31:20PM	Apr 20 1998 12:00:00.000000000AM
Dec 20 1998 3:29:55PM	Dec 21 1998 12:00:00.000000000AM
Dec 31 1998 11:59:59PM	Jan 1 1999 12:00:00.000000000AM
NULL	NULL

The following expressions round the hour portion of each date in the DATE_SHIPPED port:

```
ROUND( DATE_SHIPPED, 'HH' )
ROUND( DATE_SHIPPED, 'HH12' )
ROUND( DATE_SHIPPED, 'HH24' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:31AM	Jan 15 1998 2:00:00.000000000AM
Apr 19 1998 1:31:20PM	Apr 19 1998 2:00:00.000000000PM
Dec 20 1998 3:29:55PM	Dec 20 1998 3:00:00.000000000PM
Dec 31 1998 11:59:59PM	Jan 1 1999 12:00:00.000000000AM
NULL	NULL

The following expression rounds the minute portion of each date in the DATE_SHIPPED port:

```
ROUND( DATE_SHIPPED, 'MI' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 15 1998 2:11:00.000000000AM
Apr 19 1998 1:31:20PM	Apr 19 1998 1:31:00.000000000PM
Dec 20 1998 3:29:55PM	Dec 20 1998 3:30:00.000000000PM
Dec 31 1998 11:59:59PM	Jan 1 1999 12:00:00.000000000AM
NULL	NULL

ROUND (Numbers)

Rounds numbers to a specified number of digits or decimal places. You can also use ROUND to round dates.

Syntax

```
ROUND( numeric_value [, precision] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. You can enter any valid transformation expression. Use operators to perform arithmetic before you round the values.
precision	Optional	Positive or negative integer. If you enter a positive *precision*, the function rounds to this number of decimal places. For example, ROUND(12.99, 1) returns 13.0 and ROUND(15.44, 1) returns 15.4. If you enter a negative *precision*, the function rounds this number of digits to the left of the decimal point, returning an integer. For example, ROUND(12.99, -1) returns 10 and ROUND(15.99, -1) returns 20. If you enter decimal *precision*, the function rounds to the nearest integer before evaluating the expression. For example, ROUND(12.99, 0.8) returns 13.0 because the function rounds 0.8 to 1 and then evaluates the expression. If you omit the *precision* argument, the function rounds to the nearest integer, truncating the decimal portion of the number. For example, ROUND(12.99) returns 13.

Return Value

Numeric value.

If one of the arguments is NULL, ROUND returns NULL.

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Examples

The following expression returns the values in the Price port rounded to three decimal places:

```
ROUND( PRICE, 3 )
```

PRICE	RETURN VALUE
12.9936	12.994
15.9949	15.995
-18.8678	-18.868
56.9561	56.956
NULL	NULL

You can round digits to the left of the decimal point by passing a negative integer in the *precision* argument:

```
ROUND( PRICE, -2 )
```

PRICE	RETURN VALUE
13242.99	13200.0
1435.99	1400.0
-108.95	-100.0
NULL	NULL

If you pass a decimal value in the *precision* argument, the Data Integration Service rounds it to the nearest integer before evaluating the expression:

```
ROUND( PRICE, 0.8 )
```

PRICE	RETURN VALUE
12.99	13.0
56.34	56.3
NULL	NULL

If you omit the *precision* argument, the function rounds to the nearest integer:

```
ROUND( PRICE )
```

PRICE	RETURN VALUE
12.99	13.0
-15.99	-16.0
-18.99	-19.0

PRICE	RETURN VALUE
56.95	57.0
NULL	NULL

Tip

You can also use ROUND to explicitly set the precision of calculated values and achieve expected results. When the Data Integration Service runs in low precision mode, it truncates the result of calculations if the precision of the value exceeds 15 digits. For example, you might want to process the following expression in low precision mode:

```
7/3 * 3 = 7
```

In this case, the Data Integration Service evaluates the left hand side of the expression as 6.999999999999999 because it truncates the result of the first division operation. The Data Integration Service evaluates the entire expression as FALSE. This may not be the result you expect.

To achieve the expected result, use ROUND to round the truncated result of the left hand side of the expression to the expected result. The Data Integration Service evaluates the following expression as TRUE:

```
ROUND(7/3 * 3) = 7
```

RPAD

Converts a string to a specified length by adding blanks or characters to the end of the string.

Syntax

```
RPAD( first_string, length [,second_string] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
first_string	Required	Any string value. The strings you want to change. You can enter any valid transformation expression.
length	Required	Must be a positive integer literal. Specifies the length you want each string to be.
second_string	Optional	Any string value. Passes the string you want to append to the right-side of the *first_string* values. Enclose the characters you want to add to the end of the string within single quotation marks, for example, 'abc'. This argument is case sensitive. If you omit the second string, the function pads the end of the first string with blanks.

Return Value

String of the specified length.

NULL if a value passed to the function is NULL or if length is a negative number.

Examples

The following expression returns the item name with a length of 16 characters, appending the string '.' to the end of each item name:

```
RPAD( ITEM_NAME, 16, '.')
```

ITEM_NAME	RETURN VALUE
Flashlight	Flashlight......
Compass	Compass.........
Regulator System	Regulator System
Safety Knife	Safety Knife....

RPAD counts the length from left to right. So, if the first string is longer than the length, RPAD truncates the string from right to left. For example, RPAD('alphabetical', 5, 'x') would return the string 'alpha'. RPAD uses a partial part of the *second_string* when necessary.

The following expression returns the item name with a length of 16 characters, appending the string '*..*' to the end of each item name:

```
RPAD( ITEM_NAME, 16, '*..*' )
```

ITEM_NAME	RETURN VALUE
Flashlight	Flashlight*..**.
Compass	Compass*..**..**
Regulator System	Regulator System
Safety Knife	Safety Knife*..*

RTRIM

Removes blanks or characters from the end of a string.

If you do not specify a *trim_set* parameter in the expression:

- In UNICODE mode, RTRIM removes both single- and double-byte spaces from the end of a string.
- In ASCII mode, RTRIM removes only single-byte spaces.

If you use RTRIM to remove characters from a string, RTRIM compares the *trim_set* to each character in the *string* argument, character-by-character, starting with the right side of the string. If the character in the string matches any character in the *trim_set*, RTRIM removes it. RTRIM continues comparing and removing characters until it fails to find a matching character in the *trim_set*. It returns the string without the matching characters.

Syntax

```
RTRIM( string [, trim_set] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
string	Required	Any string value. Passes the values you want to trim. You can enter any valid transformation expression. Use operators to perform comparisons or concatenate strings before removing blanks from the end of a string.
trim_set	Optional	Any string value. Passes the characters you want to remove from the end of the string. You can also enter a text literal. However, you must enclose the characters you want to remove from the end of the string within single quotation marks, for example, 'abc'. If you omit the second string, the function removes blanks from the end of the first string. RTRIM is case sensitive.

Return Value

String. The string values with the specified characters in the *trim_set* argument removed.

NULL if a value passed to the function is NULL.

Example

The following expression removes the characters ‘re’ from the strings in the LAST_NAME port:

```
RTRIM( LAST_NAME, 're')
```

LAST_NAME	RETURN VALUE
Nelson	Nelson
Page	Pag
Osborne	Osborn
NULL	NULL
Sawyer	Sawy
H. Bender	H. Bend
Steadman	Steadman

RTRIM removes ‘e’ from Page even though ‘r’ is the first character in the *trim_set*. This is because RTRIM searches, character-by-character, for the set of characters you specify in the *trim_set* argument. If the last character in the string matches the first character in the *trim_set*, RTRIM removes it. If, however, the last character in the string does not match, RTRIM compares the second character in the *trim_set*. If the second from last character in the string matches the second character in the *trim_set*, RTRIM removes it, and so on. When the character in the string fails to match the *trim_set*, RTRIM returns the string and evaluates the next row.

In the last example, the last character in Nelson does not match any character in the *trim_set* argument, so RTRIM returns the string 'Nelson' and evaluates the next row.

Tips for RTRIM

Use RTRIM and LTRIM with || or CONCAT to remove leading and trailing blanks after you concatenate two strings.

You can also remove multiple sets of characters by nesting RTRIM. For example, if you want to remove trailing blanks and the character 't' from the end of each string in a column of names, you might create an expression similar to the following:

```
RTRIM( RTRIM( NAMES ), 't' )
```

SET_DATE_PART

Sets one part of a Date/Time value to a value you specify. With SET_DATE_PART, you can change the following parts of a date:

- **Year.** Change the year by entering a positive integer in the *value* argument. Use any of the year format strings: Y, YY, YYY, or YYYY to set the year. For example, the following expression changes the year to 2001 for all dates in the SHIP_DATE port:

  ```
  SET_DATE_PART( SHIP_DATE, 'YY', 2001 )
  ```

- **Month.** Change the month by entering a positive integer between 1 and 12 (January=1 and December=12) in the *value* argument. Use any of the month format strings: MM, MON, MONTH to set the month. For example, the following expression changes the month to October for all dates in the SHIP_DATE port:

  ```
  SET_DATE_PART( SHIP_DATE, 'MONTH', 10 )
  ```

- **Day.** Change the day by entering a positive integer between 1 and 31 (except for the months that have less than 31 days: February, April, June, September, and November) in the *value* argument. Use any of the month format strings (D, DD, DDD, DY, and DAY) to set the day. For example, the following expression changes the day to 10 for all dates in the SHIP_DATE port:

  ```
  SET_DATE_PART( SHIP_DATE, 'DD', 10 )
  ```

- **Hour.** Change the hour by entering a positive integer between 0 and 24 (where 0=12AM, 12=12PM, and 24 =12AM) in the *value* argument. Use any of the hour format strings (HH, HH12, HH24) to set the hour. For example, the following expression changes the hour to 14:00:00 (or 2:00:00PM) for all dates in the SHIP_DATE port:

  ```
  SET_DATE_PART( SHIP_DATE, 'HH', 14 )
  ```

- **Minute.** Change the minutes by entering a positive integer between 0 and 59 in the *value* argument. Use the MI format string to set the minute. For example, the following expression changes the minute to 25 for all dates in the SHIP_DATE port:

  ```
  SET_DATE_PART( SHIP_DATE, 'MI', 25 )
  ```

- **Seconds.** Change the seconds by entering a positive integer between 0 and 59 in the *value* argument. Use the SS format string to set the second. For example, the following expression changes the second to 59 for all dates in the SHIP_DATE port:

  ```
  SET_DATE_PART( SHIP_DATE, 'SS', 59 )
  ```

- **Milliseconds**. Change the milliseconds by entering a positive integer between 0 and 999 in the *value* argument. Use the MS format string to set the milliseconds. For example, the following expression changes the milliseconds to 125 for all dates in the SHIP_DATE port:

  ```
  SET_DATE_PART( SHIP_DATE, 'MS', 125 )
  ```

- **Microseconds.** Change the microseconds by entering a positive integer between 1000 and 999999 in the *value* argument. Use the US format string to set the microseconds. For example, the following expression changes the microseconds to 12555 for all dates in the SHIP_DATE port:

  ```
  SET_DATE_PART( SHIP_DATE, 'US', 12555 )
  ```

- **Nanoseconds.** Change the nanoseconds by entering a positive integer between 1000000 and 999999999 in the *value* argument. Use the NS format string to set the nanoseconds. For example, the following expression changes the nanoseconds to `12555555` for all dates in the SHIP_DATE port:

```
SET_DATE_PART( SHIP_DATE, 'NS', 12555555 )
```

Syntax

```
SET_DATE_PART( date, format, value )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. The date you want to modify. You can enter any valid transformation expression.
format	Required	Format string specifying the portion of the date to be changed. The format string is not case sensitive.
value	Required	A positive integer value assigned to the specified portion of the date. The integer must be a valid value for the part of the date you want to change. If you enter an improper value such as February 30, the session fails.

Return Value

Date in the same format as the source date with the specified part changed.

NULL if a value passed to the function is NULL.

Examples

The following expressions change the hour to 4PM for each date in the DATE_PROMISED port:

```
SET_DATE_PART( DATE_PROMISED, 'HH', 16 )
SET_DATE_PART( DATE_PROMISED, 'HH12', 16 )
SET_DATE_PART( DATE_PROMISED, 'HH24', 16 )
```

DATE_PROMISED	RETURN VALUE
Jan 1 1997 12:15:56AM	Jan 1 1997 4:15:56PM
Feb 13 1997 2:30:01AM	Feb 13 1997 4:30:01PM
Mar 31 1997 5:10:15PM	Mar 31 1997 4:10:15PM
Dec 12 1997 8:07:33AM	Dec 12 1997 4:07:33PM
NULL	NULL

The following expressions change the month to June for the dates in the DATE_PROMISED port. The Data Integration Service displays an error when you try to create a date that does not exist, such as changing March 31 to June 31:

```
SET_DATE_PART( DATE_PROMISED, 'MM', 6 )
SET_DATE_PART( DATE_PROMISED, 'MON', 6 )
SET_DATE_PART( DATE_PROMISED, 'MONTH', 6 )
```

DATE_PROMISED	RETURN VALUE
Jan 1 1997 12:15:56AM	Jun 1 1997 12:15:56AM
Feb 13 1997 2:30:01AM	Jun 13 1997 2:30:01AM
Mar 31 1997 5:10:15PM	*Error. Integration Service doesn't write row.*
Dec 12 1997 8:07:33AM	Jun 12 1997 8:07:33AM
NULL	NULL

The following expressions change the year to 2000 for the dates in the DATE_PROMISED port:

```
SET_DATE_PART( DATE_PROMISED, 'Y', 2000 )
SET_DATE_PART( DATE_PROMISED, 'YY', 2000 )
SET_DATE_PART( DATE_PROMISED, 'YYY', 2000 )
SET_DATE_PART( DATE_PROMISED, 'YYYY', 2000 )
```

DATE_PROMISED	RETURN VALUE
Jan 1 1997 12:15:56AM	Jan 1 2000 12:15:56AM
Feb 13 1997 2:30:01AM	Feb 13 2000 2:30:01AM
Mar 31 1997 5:10:15PM	Mar 31 2000 5:10:15PM
Dec 12 1997 8:07:33AM	Dec 12 2000 4:07:33PM
NULL	NULL

Tip

If you want to change multiple parts of a date at one time, you can nest multiple SET_DATE_PART functions within the *date* argument. For example, you might write the following expression to change all of the dates in the DATE_ENTERED port to July 1 1998:

```
SET_DATE_PART( SET_DATE_PART( SET_DATE_PART( DATE_ENTERED, 'YYYY', 1998),MM', 7), 'DD',
1)
```

SIGN

Returns whether a numeric value is positive, negative, or 0.

Syntax

```
SIGN( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric value. Passes the values you want to evaluate. You can enter any valid transformation expression.

Return Value

-1 for negative values.

0 for 0.

1 for positive values.

NULL if NULL.

Example

The following expression determines if the SALES port includes any negative values:

```
SIGN( SALES )
```

SALES	RETURN VALUE
100	1
-25.99	-1
0	0
NULL	NULL

SIN

Returns the sine of a numeric value (expressed in radians).

Syntax

```
SIN( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the values for which you want to calculate the sine. You can enter any valid transformation expression. You can also use operators to convert a numeric value to radians or perform arithmetic within the SIN calculation.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression converts the values in the Degrees port to radians and then calculates the sine for each radian:

```
SIN( DEGREES * 3.14159265359 / 180 )
```

DEGREES	RETURN VALUE
0	0
90	1
70	0.939692620785936
30	0.50000000000003
5	0.0871557427476639
89	0.999847695156393
NULL	NULL

You can perform arithmetic on the values passed to SIN before the function calculates the sine. For example:

```
SIN( ARCS * 3.14159265359 / 180 )
```

SINH

Returns the hyperbolic sine of the numeric value.

Syntax

```
SINH( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the values for which you want to calculate the hyperbolic sine. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the hyperbolic sine for the values in the Angles port:

```
SINH( ANGLES )
```

ANGLES	RETURN VALUE
1.0	1.1752011936438
2.897	9.03225804884884
3.66	19.4178051793031
5.45	116.376934801486
0	0.0
0.345	0.35188478309993
NULL	NULL

Tip

You can perform arithmetic on the values passed to SINH before the function calculates the hyperbolic sine. For example:

```
SINH( MEASURES.ARCS / 180 )
```

SIZE

Returns the size of the specified array or map.

Syntax

```
SIZE(value)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Array or map data type. The array or map for which you want to determine the size. If you specify an array, the function returns the number of elements in the array. If you specify a map, the function returns the number of key-value pairs in the map.

Return Value

Int.

Returns -1 if the array or map is NULL.

Examples

The following expression returns the size of the array ITEM_NAME.

```
SIZE(ITEM_NAME)
```

ITEM_NAME	RETURN VALUE
['apples','bananas','oranges']	3
['milk','coffee','tea','chai']	4
['cookie','cake']	2
['croissant',NULL]	2
NULL	-1

The following expression returns the size of the map SHIPMENT_DETAILS.

```
SIZE(SHIPMENT_DETAILS)
```

SHIPMENT_DETAILS	RETURN VALUE
[CA -> CNTR345, TX -> T234]	2
[AZ -> CNTR123, AL -> CNTR730, CA -> CNTR345]	3
[FL -> CNTR208, NULL]	2
NULL	-1

SOUNDEX

Encodes a string value into a four-character string.

SOUNDEX works for characters in the English alphabet (A-Z). It uses the first character of the input string as the first character in the return value and encodes the remaining three unique consonants as numbers.

SOUNDEX encodes characters according to the following list of rules:

- Uses the first character in *string* as the first character in the return value and encodes it in uppercase. For example, both SOUNDEX('John') and SOUNDEX('john') return 'J500'.
- Encodes the first three unique consonants following the first character in *string* and ignores the rest. For example, both SOUNDEX('JohnRB') and SOUNDEX('JohnRBCD') return 'J561'.
- Assigns a single code to consonants that sound alike.

The following table lists SOUNDEX encoding guidelines for consonants:

Table 2. SOUNDEX Encoding Guidelines for Consonants

Code	Consonant
1	B, P, F, V
2	C, S, G, J, K, Q, X, Z
3	D, T
4	L
5	M, N
6	R

- Skips the characters A, E, I, O, U, H, and W unless one of them is the first character in *string*. For example, SOUNDEX('A123') returns 'A000' and SOUNDEX('MAeiouhwC') returns 'M200'.
- If *string* produces fewer than four characters, SOUNDEX pads the resulting string with zeroes. For example, SOUNDEX('J') returns 'J000'.
- If *string* contains a set of consecutive consonants that use the same code listed in "SOUNDEX" on page 188, SOUNDEX encodes the first occurrence and skips the remaining occurrences in the set. For example, SOUNDEX('AbbpdMN') returns 'A135'.
- Skips numbers in *string*. For example, both SOUNDEX('Joh12n') and SOUNDEX('1John') return 'J500'.
- Returns NULL if *string* is NULL or if all the characters in *string* are not letters of the English alphabet.

Syntax

```
SOUNDEX( string )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
string	Required	Character string. Passes the string value you want to encode. You can enter any valid transformation expression.

Return Value

String.

NULL if one of the following conditions is true:

- If value passed to the function is NULL.
- No character in *string* is a letter of the English alphabet.
- *string* is empty.

Example

The following expression encodes the values in the EMPLOYEE_NAME port:

```
SOUNDEX( EMPLOYEE_NAME )
```

EMPLOYEE_NAME	RETURN VALUE
John	J500
William	W450
jane	J500
joh12n	J500
1abc	A120
NULL	NULL

SQL_LIKE

Returns whether a value matches a regular expression pattern. This lets you validate date patterns, such as IDs, telephone numbers, postal codes, and state names.

Syntax

```
SQL_LIKE(subject, pattern, escape character)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
subject	Required	String data type. Passes the value you want to match against the regular expression. Enclose the value in single quotation marks.
pattern	Required	String data type. Regular expression that you want to match. Enclose the pattern in single quotation marks.
escape character	Optional	String data type. The SQL_LIKE function supports the percentage sign (%) and underscore (_) as escape characters. Enclose the escape character in single quotation marks.

Return Value

TRUE if the data matches the pattern.

FALSE if the data does not match the pattern.

NULL if the input is a null value or if the pattern is NULL.

Example

You might use SQL_LIKE in an expression to find names that match a pattern. For example, the following expression matches names against the pattern "A_#%" with the escape character '#':

```
SQL_LIKE(ENAME, 'A_#%', '#')
```

ENAME	Value
SMITH	FALSE
AX%	TRUE
MILLER	FALSE
A%	FALSE
JONES	FALSE
BLAKE	FALSE
A%1	FALSE

SQRT

Returns the square root of a non-negative numeric value.

Syntax

```
SQRT( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Positive numeric value. Passes the values for which you want to calculate a square root. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the square root for the values in the Numbers port:

```
SQRT( NUMBERS )
```

NUMBERS	RETURN VALUE
100	10
-100	*Error. Data Integration Service does not write row.*

NUMBERS	RETURN VALUE
NULL	NULL
60.54	7.78074546557076

The value -100 results in an error, since the function SQRT only evaluates positive numeric values. If you pass a negative value or character value, the Data Integration Service displays a Transformation Evaluation Error and does not write the row.

You can perform arithmetic on the values passed to SQRT before the function calculates the square root.

STDDEV

Returns the standard deviation of the numeric values you pass to this function. STDDEV is used to analyze statistical data. You can nest only one other aggregate function within STDDEV, and the nested function must return a Numeric datatype.

Syntax

```
STDDEV( numeric_value [,filter_condition] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatypes. This function passes the values for which you want to calculate a standard deviation or the results of a function. You can enter any valid transformation expression. You can use operators to average values in different ports.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Nulls

If a single value is NULL, STDDEV ignores it. However, if all values are NULL, STDDEV returns NULL.

Group By

STDDEV groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, STDDEV treats all rows as one group, returning one value.

Examples

The following expression calculates the standard deviation of all rows greater than $2000.00 in the TOTAL_SALES port:

```
STDDEV( SALES, SALES > 2000.00 )
```

SALES
2198.0
1010.90
2256.0
153.88
3001.0
NULL
8953.0

RETURN VALUE: 3254.60361129688

The function does not include the values 1010.90 and 153.88 in the calculation because the *filter_condition* specifies sales greater than $2,000.

The following expression calculates the standard deviation of all rows in the SALES port:

```
STDDEV(SALES)
```

SALES
2198.0
2198.0
2198.0
2198.0

RETURN VALUE: 0

The return value is 0 because each row contains the same number (no standard deviation exists). If there is no standard deviation, the return value is 0.

STRUCT

Generates a struct with element names and data types based on the specified arguments.

Syntax

```
STRUCT(element_name1, value1 as any [, element_name2, value2 as any] ...)
```

The following table describes the arguments for this command:

Argument	Required/Optional	Description
element_name1	Required	The name of the struct element.
value1	Required	Any data type. The value of the struct element.

If you use the STRUCT function in an output expression for a struct port, the function arguments must match the data type of the elements in the complex data type definition.

Return Value

Struct.

Examples

The following expression generates a struct.

```
STRUCT(city , 'New York', state, 'NY')
```

RETURN VALUE

```
{
    city:New York
    state:NY
}
```

The following expression generates a struct for a struct output port with a complex data type definition **adrs_typedef**:

```
STRUCT(city, cust_city, state, cust_state)
```

The complex data type definition **adrs_typedef** is defined in the type definition library as follows:

```
adrs_typedef{
    city : string
    state : string
}
```

cust_city	cust_state	RETURN VALUE

NEWYORK	NY	{
		city:NEWYORK
		state:NY
		}

REDWOOD CITY	CA	{
		city:REDWOOD CITY
		state:CA
		}

STRUCT_AS

Generates a struct with a schema based on the specified complex data type definition and the values you pass as argument.

Syntax

```
STRUCT_AS (:Type.type_definition_library.type_definition, struct_value)
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
:Type.type_definition_library.type_definition	Required	The complex data type definition that represents the schema of the struct data. Use the reference qualifier `:Type` to reference the type definition library that contains the complex data type definition.
struct_value	Required	Value for each element in the complex data type definition separated by comma.

Return Value

Struct.

Examples

The following expression generates a struct based on the specified complex data type definition h1_address_def with the values that you pass as arguments for the struct elements.

```
STRUCT_AS (:Type.type_definition_library.h1_address_def, City, State, ZIP)
```

The complex data type definition **h1_address_def** is defined in the type definition library as follows:

```
h1_address_def{
    city : string
    state : string
    zip : int
}
```

city	state	zip	RETURN VALUE

NEWYORK	NY	12345	{
			city:NEWYORK
			state:NY
			zip:12345
			}

REDWOOD CITY	CA	23452	{
			city:REDWOOD CITY
			state:CA
			zip:23452
			}

SUBSTR

Returns a portion of a string. SUBSTR counts all characters, including blanks, starting at the beginning of the string.

Syntax

```
SUBSTR( string, start [,length] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
string	Required	Must be a character string. Passes the strings you want to search. You can enter any valid transformation expression. If you pass a numeric value, the function converts it to a character string.
start	Required	Must be an integer. The position in the string where you want to start counting. You can enter any valid transformation expression. If the start position is a positive number, SUBSTR locates the start position by counting from the beginning of the string. If the start position is a negative number, SUBSTR locates the start position by counting from the end of the string. If the start position is 0, SUBSTR searches from the first character in the string.
length	Optional	Must be an integer greater than 0. The number of characters you want SUBSTR to return. You can enter any valid transformation expression. If you omit the length argument, SUBSTR returns all of the characters from the start position to the end of the string. If you pass a negative integer or 0, the function returns an empty string. If you pass a decimal, the function rounds it to the nearest integer value.

Return Value

String.

Empty string if you pass a negative or 0 length value.

NULL if a value passed to the function is NULL.

Examples

The following expressions return the area code for each row in the Phone port:

```
SUBSTR( PHONE, 0, 3 )
```

PHONE	RETURN VALUE
809-555-0269	809
357-687-6708	357
NULL	NULL

```
SUBSTR( PHONE, 1, 3 )
```

PHONE	RETURN VALUE
809-555-3915	809

PHONE	RETURN VALUE
357-687-6708	357
NULL	NULL

The following expressions return the phone number without the area code for each row in the Phone port:

```
SUBSTR( PHONE, 5, 8 )
```

PHONE	RETURN VALUE
808-555-0269	555-0269
809-555-3915	555-3915
357-687-6708	687-6708
NULL	NULL

You can also pass a negative start value to return the phone number for each row in the Phone port. The expression still reads the source string from left to right when returning the result of the *length* argument:

```
SUBSTR( PHONE, -8, 3 )
```

PHONE	RETURN VALUE
808-555-0269	555
809-555-3915	555
357-687-6708	687
NULL	NULL

You can nest INSTR in the *start* or *length* argument to search for a specific string and return its position.

The following expression evaluates a string, starting from the end of the string. The expression finds the last (right-most) space in the string and then returns all characters preceding it:

```
SUBSTR( CUST_NAME,1,INSTR( CUST_NAME,' ' ,-1,1 ) - 1 )
```

CUST_NAME	RETURN VALUE
PATRICIA JONES	PATRICIA
MARY ELLEN SHAH	MARY ELLEN

The following expression removes the character '#' from a string:

```
SUBSTR( CUST_ID, 1, INSTR(CUST_ID, '#')-1 ) || SUBSTR( CUST_ID, INSTR(CUST_ID, '#')+1 )
```

When the *length* argument is longer than the string, SUBSTR returns all the characters from the start position to the end of the string. Consider the following example:

```
SUBSTR('abcd', 2, 8)
```

The return value is ‘bcd’. Compare this result to the following example:

```
SUBSTR('abcd', -2, 8)
```

The return value is ‘cd’.

SUM

Returns the sum of all values in the selected port. Optionally, you can apply a filter to limit the rows you read to calculate the total. You can nest only one other aggregate function within SUM, and the nested function must return a Numeric datatype.

Syntax

```
SUM( numeric_value [, filter_condition ] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values you want to add. You can enter any valid transformation expression. You can use operators to add values in different ports.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Numeric value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the filter condition evaluates to FALSE or NULL for all rows).

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Nulls

If a single value is NULL, SUM ignores it. However, if all values passed from the port are NULL, SUM returns NULL.

Group By

SUM groups values based on group by ports you define in the transformation, returning one result for each group.

If there is no group by port, SUM treats all rows as one group, returning one value.

Example

The following expression returns the sum of all values greater than 2000 in the Sales port:

```
SUM( SALES, SALES > 2000 )
```

```
SALES
2500.0
1900.0
1200.0
NULL
3458.0
4519.0
```

RETURN VALUE: 10477.0

Tip

You can perform arithmetic on the values passed to SUM before the function calculates the total. For example:

```
SUM( QTY * PRICE - DISCOUNT )
```

SYSTIMESTAMP

Returns the current date and time of the node hosting the Data Integration Service with precision to the nanosecond. The precision to which you display the date and time depends on the platform.

The return value of the function varies depending on how you configure the argument:

- When you configure the argument of SYSTIMESTAMP as a variable, the Data Integration Service evaluates the function for each row in the transformation.
- When you configure the argument of SYSTIMESTAMP as a constant, the Data Integration Service evaluates the function once and retains the value for each row in the transformation.

Syntax

```
SYSTIMESTAMP( [format] )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
format	Optional	Precision to which you want to retrieve the timestamp. You can specify precision up to seconds (SS), milliseconds (MS), microseconds (US), or nanoseconds (NS). Enclose the format string within single quotation marks. The format string is not case sensitive. For example, to display the date and time to the precision of milliseconds use the following syntax: SYSTIMESTAMP('MS'). Default precision is microseconds (US).

Return Value

Timestamp. Returns date and time to the specified precision.

Examples

Your organization has an online order service and processes real-time data. You can use the SYSTIMESTAMP function to generate a primary key for each transaction in the target database.

Create an Expression transformation with the following ports and values:

Port Name	Port Type	Expression
Customer_Name	Input	n/a
Order_Qty	Input	n/a
Time_Counter	Variable	'US'
Transaction_Id	Output	SYSTIMESTAMP (Time_Counter)

At run time, the Data Integration Service generates the system time to the precision of microseconds for each row:

Customer_Name	Order_Qty	Transaction_Id
Vani Deed	14	07/06/2007 18:00:30.701015000
Kalia Crop	3	07/06/2007 18:00:30.701029000
Vani Deed	6	07/06/2007 18:00:30.701039000
Harry Spoon	32	07/06/2007 18:00:30.701048000

TAN

Returns the tangent of a numeric value (expressed in radians).

Syntax

```
TAN( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the numeric values for which you want to calculate the tangent. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the tangent for all values in the Degrees port:

```
TAN( DEGREES * 3.14159 / 180 )
```

DEGREES	RETURN VALUE
70	2.74747741945531
50	1.19175359259435
30	0.577350269189672
5	0.0874886635259298
18	0.324919696232929
89	57.2899616310952
NULL	NULL

TANH

Returns the hyperbolic tangent of the numeric value passed to this function.

Syntax

```
TANH( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Numeric data expressed in radians (degrees multiplied by pi divided by 180). Passes the numeric values for which you want to calculate the hyperbolic tangent. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to the function is NULL.

Example

The following expression returns the hyperbolic tangent for the values in the Angles port:

```
TANH( ANGLES )
```

ANGLES	RETURN VALUE
1.0	0.761594155955765
2.897	0.993926947790665
3.66	0.998676551914886
5.45	0.999963084213409
0	0.0
0.345	0.331933853503641
NULL	NULL

Tip

You can perform arithmetic on the values passed to TANH before the function calculates the hyperbolic tangent. For example:

```
TANH( ARCS / 360 )
```

TIME_RANGE

Determines the time range for the streaming events to be joined.

The `TIME_RANGE` function is applicable only for a Joiner transformation in a streaming mapping.

Syntax

```
TIME_RANGE(EventTime1,EventTime2,Format,Interval)
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
EventTime1	Required	Date datatype. The time that a streaming event is generated in the master port of a Joiner transformation.
EventTime2	Required	Date datatype. The time that a streaming event is generated in the detail port of a Joiner transformation.

Argument	Required/ Optional	Description
Format	Required	A format string that specifies the portion of the event time value you want to change. Enclose the format string within single quotation marks. For example, 'Seconds'. The format string is not case-sensitive. The format argument accepts the following values: - Years - Months - Weeks - Days - Hours - Minutes - Seconds - Milliseconds - Microseconds
Interval	Required	An integer value which you want to change the event time value based on the format.

Return Value

NULL if you pass a null value to the function.

Example

The following example returns the time range expression for the Joiner transformation:

```
TIME_RANGE(EventTime1,EventTime2,'Second',4)
```

RETURN VALUE:

```
(EventTime1.<=(EventTime2).&&(EventTime2.<=(EventTime1.+(expr("INTERVAL 4 SECONDS")))))
```

TO_BIGINT

Converts a string or numeric value to a bigint value. TO_BIGINT syntax contains an optional argument that you can choose to round the number to the nearest integer or truncate the decimal portion. TO_BIGINT ignores leading blanks.

Syntax

```
TO_BIGINT( value [, flag] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	String or numeric datatype. Passes the value you want to convert to a bigint value. You can enter any valid transformation expression.
flag	Optional	Specifies whether to truncate or round the decimal portion. The flag must be an integer literal or the constants TRUE or FALSE. TO_BIGINT truncates the decimal portion when the flag is TRUE or a number other than 0. TO_BIGINT rounds the value to the nearest integer if the flag is FALSE or 0 or if you omit this argument. The flag is not set by default.

Return Value

Bigint.

NULL if the value passed to the function is NULL.

If the value passed to the function contains data that is not valid for a bigint value, the Data Integration Service marks the row as an error row or fails the mapping.

Examples

The following expressions use values from the port IN_TAX:

```
TO_BIGINT( IN_TAX, TRUE )
```

IN_TAX	RETURN VALUE
'7245176201123435.6789'	7245176201123435
'7245176201123435.2'	7245176201123435
'7245176201123435.2.48'	7245176201123435
NULL	NULL
'A12.3Grove'	*Error. Integration Service skips this row.*
' 176201123435.87'	176201123435
'-7245176201123435.2'	-7245176201123435
'-7245176201123435.23'	-7245176201123435
-9223372036854775806.9	-9223372036854775806
9223372036854775806.9	9223372036854775806

```
TO_BIGINT( IN_TAX )
```

IN_TAX	RETURN VALUE
'7245176201123435.6789'	7245176201123436

IN_TAX	RETURN VALUE
'7245176201123435.2'	7245176201123435
'7245176201123435.348'	7245176201123435
NULL	NULL
'A12.3Grove'	*Error. Integration Service skips this row.*
' 176201123435.87'	176201123436
'-7245176201123435.6789'	-7245176201123436
'-7245176201123435.23'	-7245176201123435
-9223372036854775806.9	-9223372036854775807
9223372036854775806.9	9223372036854775807

TO_CHAR (Dates)

Converts dates to character strings. TO_CHAR also converts numeric values to strings. You can convert the date into any format using the TO_CHAR format strings.

TO_CHAR (date [,format]) converts a data type or internal value of date, Timestamp, Timestamp with Time Zone, or Timestamp with Local Time Zone data type to a value of string data type specified by the format string.

Syntax

```
TO_CHAR( date [,format] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. Passes the date values you want to convert to character strings. You can enter any valid transformation expression.
format	Optional	Enter a valid TO_CHAR format string. The format string defines the format of the return value, not the format for the values in the date argument. If you omit the format string, the function returns a string based on the date format specified in the mapping configuration.

Return Value

String.

NULL if a value passed to the function is NULL.

Examples

The following expression converts the dates in the DATE_PROMISED port to text in the format MON DD YYYY:

```
TO_CHAR( DATE_PROMISED, 'MON DD YYYY' )
```

DATE_PROMISED	RETURN VALUE
Apr 1 1998 12:00:10AM	'Apr 01 1998'
Feb 22 1998 01:31:10PM	'Feb 22 1998'
Oct 24 1998 02:12:30PM	'Oct 24 1998'
NULL	NULL

If you omit the *format* argument, TO_CHAR returns a string in the date format specified in the mapping configuration, by default, MM/DD/YYYY HH24:MI:SS.US:

```
TO_CHAR( DATE_PROMISED )
```

DATE_PROMISED	RETURN VALUE
Apr 1 1998 12:00:10AM	'04/01/1998 00:00:10.000000'
Feb 22 1998 01:31:10PM	'02/22/1998 13:31:10.000000'
Oct 24 1998 02:12:30PM	'10/24/1998 14:12:30.000000'
NULL	NULL

The following expressions return the day of the week for each date in a port:

```
TO_CHAR( DATE_PROMISED, 'D' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'3'
02-22-1997 01:31:10PM	'7'
10-24-1997 02:12:30PM	'6'
NULL	NULL

```
TO_CHAR( DATE_PROMISED, 'DAY' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'Tuesday'
02-22-1997 01:31:10PM	'Saturday'

DATE_PROMISED	RETURN VALUE
10-24-1997 02:12:30PM	'Friday'
NULL	NULL

The following expression returns the day of the month for each date in a port:

```
TO_CHAR( DATE_PROMISED, 'DD' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'01'
02-22-1997 01:31:10PM	'22'
10-24-1997 02:12:30PM	'24'
NULL	NULL

The following expression returns the day of the year for each date in a port:

```
TO_CHAR( DATE_PROMISED, 'DDD' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'091'
02-22-1997 01:31:10PM	'053'
10-24-1997 02:12:30PM	'297'
NULL	NULL

The following expressions return the hour of the day for each date in a port:

```
TO_CHAR( DATE_PROMISED, 'HH' )
TO_CHAR( DATE_PROMISED, 'HH12' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'12'
02-22-1997 01:31:10PM	'01'
10-24-1997 02:12:30PM	'02'
NULL	NULL

```
TO_CHAR( DATE_PROMISED, 'HH24' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'00'

DATE_PROMISED	RETURN VALUE
02-22-1997 01:31:10PM	'13'
10-24-1997 11:12:30PM	'23'
NULL	NULL

The following expression converts date values to MJD values expressed as strings:

```
TO_CHAR( SHIP_DATE, 'J')
```

SHIP_DATE	RETURN_VALUE
Dec 31 1999 03:59:59PM	2451544
Jan 1 1900 01:02:03AM	2415021

The following expression converts dates to strings in the format MM/DD/YY:

```
TO_CHAR( SHIP_DATE, 'MM/DD/RR')
```

SHIP_DATE	RETURN_VALUE
12/31/1999 01:02:03AM	12/31/99
09/15/1996 03:59:59PM	09/15/96
05/17/2003 12:13:14AM	05/17/03

You can also use the format string SSSSS in a TO_CHAR expression. For example, the following expression converts the dates in the SHIP_DATE port to strings representing the total seconds since midnight:

```
TO_CHAR( SHIP_DATE, 'SSSSS')
```

SHIP_DATE	RETURN_VALUE
12/31/1999 01:02:03AM	3783
09/15/1996 03:59:59PM	86399

In TO_CHAR expressions, the YY format string produces the same results as the RR format string.

The following expression converts dates to strings in the format MM/DD/YY:

```
TO_CHAR( SHIP_DATE, 'MM/DD/YY')
```

SHIP_DATE	RETURN_VALUE
12/31/1999 01:02:03AM	12/31/99
09/15/1996 03:59:59PM	09/15/96
05/17/2003 12:13:14AM	05/17/03

The following expression returns the week of the month for each date in a port:

```
TO_CHAR( DATE_PROMISED, 'W' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10AM	'01'
02-22-1997 01:31:10AM	'04'
10-24-1997 02:12:30PM	'04'
NULL	NULL

The following expression returns the week of the year for each date in a port:

```
TO_CHAR( DATE_PROMISED, 'WW' )
```

DATE_PROMISED	RETURN VALUE
04-01-1997 12:00:10PM	'18'
02-22-1997 01:31:10AM	'08'
10-24-1997 02:12:30AM	'43'
NULL	NULL

Tip

You can combine TO_CHAR and TO_DATE to convert a numeric value for a month into the text value for a month using a function such as:

```
TO_CHAR( TO_DATE( numeric_month, 'MM' ), 'MONTH' )
```

TO_CHAR (Numbers)

Converts numeric values to text strings. TO_CHAR also converts dates to strings.

Syntax

```
TO_CHAR( numeric_value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric data type. The numeric value you want to convert to a string. You can enter any valid transformation expression.

TO_CHAR converts double values to text strings as follows:

- Converts double values of up to 16 digits to strings and provides accuracy up to 15 digits. If you pass a number with more than 15 digits, TO_CHAR rounds the number based on the sixteenth digit and returns the string representation of the number in scientific notation. For example, 1234567890123456 double value converts to '1.23456789012346e+015' string value.
- Returns decimal notation for numbers in the ranges (-1e16,-1e-16] and [1e-16, 1e16). TO_CHAR returns scientific notation for numbers outside these ranges. For example, 10842764968208837340 double value converts to '1.08427649682088e+019' string value.

TO_CHAR converts decimal values to text strings as follows:

- In high precision mode, TO_CHAR converts decimal values of up to 38 digits to strings. If you pass a decimal value with more than 38 digits, TO_CHAR returns scientific notation for numbers greater than 38 digits.
- In low precision mode, TO_CHAR treats decimal values as double values.
- If you pass a decimal port to the TO_CHAR function and the input value does not have enough digits to match the scale of the decimal port, the TO_CHAR function appends zeros to the value.

  For example, if the scale of the decimal port is 5 and the value in a row is 7.6901, the TO_CHAR function treats the input value as 7.69010 and the return value is '7.69010.'

Return Value

String.

NULL if a value passed to the function is NULL.

Double Conversion Example

The following expression converts the double values in the SALES port to strings:

```
TO_CHAR( SALES )
```

SALES	RETURN VALUE
1010.99	'1010.99'
-15.62567	'-15.62567'
10842764968208837340	'1.08427649682088e+019' *(rounded based on the 16th digit and returns the value in scientific notation)*
236789034569723	'236789034569723'
0	'0'
33.15	'33.15'
NULL	NULL

Decimal Conversion Example

The following expression converts the decimal values in the SALES port to strings in high precision mode:

```
TO_CHAR( SALES )
```

SALES	RETURN VALUE
2378964536789761	'2378964536789761'

SALES	RETURN VALUE
12345678901234567890123456789	'12345678901234567890123456789'
1.23457894546964934587612345 6	'1.23457894546964934587612345 6'
0.99999999999999999999999999	'0.99999999999999999999999999'
12345678901234567890123456799	'12345678901234567890123456799'
23456788992233456678458934567123465239	'23456788992233456678458934567123465239'
42345678901234567890123456799123456789 9 *(greater than 38)*	'4.23456789012346e+038'

TO_DATE

Converts a character string to a Date/Time datatype. You use the TO_DATE format strings to specify the format of the source strings.

The output port must be Date/Time for TO_DATE expressions.

If you are converting two-digit years with TO_DATE, use either the RR or YY format string. Do not use the YYYY format string.

Syntax

```
TO_DATE( string [, format] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
string	Required	Must be a string datatype. Passes the values that you want to convert to dates. You can enter any valid transformation expression.
format	Optional	Enter a valid TO_DATE format string. The format string must match the parts of the *string* argument. For example, if you pass the string 'Mar 15 1998 12:43:10AM', you must use the format string 'MON DD YYYY HH12:MI:SSAM'. If you omit the format string, the string value must be in the date format specified in the session.

Return Value

Date.

TO_DATE always returns a date and time. If you pass a string that does not have a time value, the date returned always includes the time 00:00:00.000000000. You can map the results of this function to any target column with a datetime datatype. If the target column precision is less than nanoseconds, the Data Integration Service truncates the datetime value to match the precision of the target column when it writes datetime values to the target.

NULL if you pass a null value to this function.

Warning: The format of the TO_DATE string must match the format string including any date separators. If it does not, the Data Integration Service might return inaccurate values or skip the record.

Examples

The following expression returns date values for the strings in the DATE_PROMISED port. TO_DATE always returns a date and time. If you pass a string that does not have a time value, the date returned always includes the time 00:00:00.000000000. If you run a mapping in the twentieth century, the century will be 19. In this example, the current year on the node running the Data Integration Service is 1998. The datetime format for the target column is MON DD YY HH24:MI SS, so the Data Integration Service truncates the datetime value to seconds when it writes to the target:

```
TO_DATE( DATE_PROMISED, 'MM/DD/YY' )
```

DATE_PROMISED	RETURN VALUE
'01/22/98'	Jan 22 1998 00:00:00
'05/03/98'	May 3 1998 00:00:00
'11/10/98'	Nov 10 1998 00:00:00
'10/19/98'	Oct 19 1998 00:00:00
NULL	NULL

The following expression returns date and time values for the strings in the DATE_PROMISED port. If you pass a string that does not have a time value, the Data Integration Service returns an error. If you run a mapping in the twentieth century, the century will be 19. The current year on the node running the Data Integration Service is 1998:

```
TO_DATE( DATE_PROMISED, 'MON DD YYYY HH12:MI:SSAM' )
```

DATE_PROMISED	RETURN VALUE
'Jan 22 1998 02:14:56PM'	Jan 22 1998 02:14:56PM
'Mar 15 1998 11:11:11AM'	Mar 15 1998 11:11:11AM
'Jun 18 1998 10:10:10PM'	Jun 18 1998 10:10:10PM
'October 19 1998'	*Error. Integration Service skips this row.*
NULL	NULL

The following expression converts strings in the SHIP_DATE_MJD_STRING port to date values:

```
TO_DATE (SHIP_DATE_MJD_STR, 'J')
```

SHIP_DATE_MJD_STR	RETURN_VALUE
'2451544'	Dec 31 1999 00:00:00.000000000
'2415021'	Jan 1 1900 00:00:00.000000000

Because the J format string does not include the time portion of a date, the return values have the time set to 00:00:00.000000000.

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/RR')
```

DATE_STR	RETURN VALUE
'04/01/98'	04/01/1998 00:00:00.000000000
'08/17/05'	08/17/2005 00:00:00.000000000

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/YY')
```

DATE_STR	RETURN VALUE
'04/01/98'	04/01/1998 00:00:00.000000000
'08/17/05'	08/17/1905 00:00:00.000000000

Note: For the second row, RR returns the year 2005 and YY returns the year 1905.

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/Y')
```

DATE_STR	RETURN VALUE
'04/01/8'	04/01/1998 00:00:00.000000000
'08/17/5'	08/17/1995 00:00:00.000000000

The following expression converts a string to a four-digit year format. The current year is 1998:

```
TO_DATE( DATE_STR, 'MM/DD/YYY')
```

DATE_STR	RETURN VALUE
'04/01/998'	04/01/1998 00:00:00.000000000
'08/17/995'	08/17/1995 00:00:00.000000000

The following expression converts strings that includes the seconds since midnight to date values:

```
TO_DATE( DATE_STR, 'MM/DD/YYYY SSSSS')
```

DATE_STR	RETURN_VALUE
'12/31/1999 3783'	12/31/1999 01:02:03
'09/15/1996 86399'	09/15/1996 23:59:59

If the target accepts different date formats, use TO_DATE and IS_DATE with the DECODE function to test for acceptable formats. For example:

```
DECODE( TRUE,

 --test first format
```

```
  IS_DATE( CLOSE_DATE,'MM/DD/YYYY HH24:MI:SS' ),

--if true, convert to date
     TO_DATE( CLOSE_DATE,'MM/DD/YYYY HH24:MI:SS' ),

--test second format; if true, convert to date
  IS_DATE( CLOSE_DATE,'MM/DD/YYYY'), TO_DATE( CLOSE_DATE,'MM/DD/YYYY' ),

--test third format; if true, convert to date
  IS_DATE( CLOSE_DATE,'MON DD YYYY'), TO_DATE( CLOSE_DATE,'MON DD YYYY'),

--if none of the above
  ERROR( 'NOT A VALID DATE') )
```

You can combine TO_CHAR and TO_DATE to convert a numeric value for a month into the text value for a month using a function such as:

```
TO_CHAR( TO_DATE( numeric_month, 'MM' ), 'MONTH' )
```

TO_DECIMAL

Converts a string or numeric value to a decimal value. TO_DECIMAL ignores leading blanks.

Syntax

```
TO_DECIMAL( value [, scale] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Must be a string or numeric datatype. Passes the values you want to convert to decimal values. You can enter any valid transformation expression.
scale	Optional	Must be an integer literal between 0 and 28, inclusive. Specifies the number of digits allowed after the decimal point. If you omit this argument, the function returns a value with the same scale as the input value. If you pass a decimal to the TO_DECIMAL function to cast the decimal to a decimal with a different scale, the scale argument is a maximum and cannot extend scale. For example, if the scale argument is 5 and the scale of the input value is 6, the fractional digits are truncated. If the scale of the input value is 4, the fractional digits remain the same.

Return Value

Decimal of precision and scale between 0 and 28, inclusive.

NULL if a value passed to the function is NULL.

If the value passed to the function contains data that is not valid for a decimal value, the Data Integration Service marks the row as an error row.

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 28 digits.

Example

This expression uses values from the port IN_TAX. IN_TAX is a String data type with precision of 44 digits. RETURN VALUE is a Decimal data type with a precision of 28 and scale of 3:

```
TO_DECIMAL( IN_TAX, 3 )
```

IN_TAX	RETURN VALUE
'15.6789'	15.679
'60.2'	60.200
'118.348'	118.348
NULL	NULL
'A12.3Grove'	*Error. Integration Service skips this row.*
'711A1'	*Error. Integration Service skips this row.*
'1234567890.123'	1234567890.123
'12345678901234567890123456789012345678901234567890.123'	Error. Integration Service skips this row.
'12345678901234567890123456789012345678901234567890.123	Error. Integration Service skips this row.

TO_DECIMAL38

Converts a string or numeric value to a decimal value. TO_DECIMAL38 ignores leading blanks.

Syntax

```
TO_DECIMAL38( value [, scale] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	Must be a string or numeric data type. Passes the values that you want to convert to decimal values. You can enter any valid transformation expression.
scale	Optional	Must be an integer literal between 0 and 38, inclusive. Specifies the number of digits allowed after the decimal point. If you omit this argument, the function returns a value with the same scale as the input value. If you pass a decimal to the TO_DECIMAL38 function to cast the decimal to a decimal with a different scale, the scale argument is a maximum and cannot extend scale. For example, if the scale argument is 5 and the scale of the input value is 6, the fractional digits are truncated. If the scale of the input value is 4, the fractional digits remain the same.

Return Value

Decimal of precision and scale between 0 and 38, inclusive.

NULL if a value passed to the function is NULL.

If the value passed to the function contains data that is not valid for a decimal value, the Data Integration Service marks the row as an error row. For example, if you pass `TO_DECIMAL38("12345678901234567890123456789012345678901234567890.12")`, the Data Integration Service rejects the row.

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Example

This expression uses values from the port IN_TAX. IN_TAX is a String data type with precision of 44 digits. RETURN VALUE is a Decimal data type with a precision of 38 and a scale of 3:

```
TO_DECIMAL38( IN_TAX, 3 )
```

IN_TAX	RETURN VALUE
'15.6789'	15.679
'60.2'	60.200
'118.348'	118.348
NULL	NULL
'A12.3Grove'	*Error. Integration Service skips this row.*
'1234567890.123'	1234567890.123
'12345678901234567890123456789012345678901234567890.123'	12345678901234567890123456789012345678901234567890.123
'12345678901234567890123456789012345678901234567890.123'	*Error. Integration Service skips this row.*
'711A1'	*Error. Integration Service skips this row.*

TO_FLOAT

Converts a string or numeric value to a double-precision floating point number (the Double datatype). TO_FLOAT ignores leading blanks.

Syntax

```
TO_FLOAT( value )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
value	Required	Must be a string or numeric datatype. Passes the values you want to convert to double values. You can enter any valid transformation expression.

Return Value

Double value.

NULL if a value passed to this function is NULL.

If the value passed to the function contains data that is not valid for a float value, the Data Integration Service marks the row as an error row or fails the mapping.

Example

This expression uses values from the port IN_TAX:

```
TO_FLOAT( IN_TAX )
```

IN_TAX	RETURN VALUE
'15.6789'	15.6789
'60.2'	60.2
'118.348'	118.348
NULL	NULL
'A12.3Grove'	*Error. Integration Service skips this row.*

TO_INTEGER

Converts a string or numeric value to an integer. TO_INTEGER syntax contains an optional argument that you can choose to round the number to the nearest integer or truncate the decimal portion. TO_INTEGER ignores leading blanks.

Syntax

```
TO_INTEGER( value [, flag] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
value	Required	String or numeric datatype. Passes the value you want to convert to an integer. You can enter any valid transformation expression.
flag	Optional	Specifies whether to truncate or round the decimal portion. The flag must be an integer literal or the constants TRUE or FALSE. TO_INTEGER truncates the decimal portion when the flag is TRUE or a number other than 0. TO_INTEGER rounds the value to the nearest integer if the flag is FALSE or 0 or if you omit this argument.

Return Value

Integer.

NULL if the value passed to the function is NULL.

If the value passed to the function contains data that is not valid for an integer value, the Data Integration Service marks the row as an error row or fails the mapping.

Examples

The following expressions use values from the port IN_TAX. The Data Integration Service displays an error when the conversion causes a numeric overflow:

```
TO_INTEGER( IN_TAX, TRUE )
```

IN_TAX	RETURN VALUE
'15.6789'	15
'60.2'	60
'118.348'	118
'5,000,000,000'	*Error. Integration Service skips this row.*
NULL	NULL
'A12.3Grove'	*Error. Integration Service skips this row.*
' 123.87'	123
'-15.6789'	-15
'-15.23'	-15

```
TO_INTEGER( IN_TAX, FALSE)
```

IN_TAX	RETURN VALUE
'15.6789'	16
'60.2'	60

IN_TAX	RETURN VALUE
'118.348'	118
'5,000,000,000'	*Error. Integration Service skips this row.*
NULL	NULL
'A12.3Grove'	*Error. Integration Service skips this row.*
' 123.87'	124
'-15.6789'	-16
'-15.23'	-15

TO_TIMESTAMP_TZ

Converts a string to Timestamp with Time Zone value. The function returns Timestamp with Time Zone data type. You use the TO_TIMESTAMP_TZ format strings to specify the format of the source strings.

Syntax

```
TO_TIMESTAMP_TZ ( String , [format] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
String	Required	Must be a string data type. Passes the values you want to convert to Timestamp with Time Zone. You can enter any valid transformation expression. The string must be a character string.
format	Optional	Enter a valid TO_TIMESTAMP_TZ format string. The format string must match the parts of the string argument. For example, if you pass the string 'Mar 15 1997 12:43:10AM ASIA/CALCUTTA', you must use the format string 'MON DD YYYY HH12:MI:SSAM TZR'. If you do not specify the format string, the function uses the default date time format in the Run Configurations dialog.

Return Value

Returns a timestamp with time zone data type.

NULL if the input is a null value.

If the value passed to the function contains data that is not valid for a timestamp with time zone value, the Data Integration Service marks the row as an error row or fails the mapping.

Example

INPUT VALUE	RETURN VALUE
‘1947-08-05 10:45:00.221111000 AM America/Los_Angeles’, 'YYYY-MM-DD HH:MI:SS.NS AM TZR'	Returns a timestamp with time zone data type with the following data: ‘1947-08-05 10:45:00.221111000 AM AMERICA/LOS_ANGELES’
‘1947-08-05 10:45:00.221111000 AM America/Los_Angeles’, 'YYYY-MM-DD HH:MI:SS.NS AM'	Returns a timestamp with time zone data type even without specifying time zone region in the time zone region format: ‘1947-08-05 10:45:00.221111000 AM AMERICA/LOS_ANGELES'
‘1947-08-05 10:45:00.221111000 AM America/Los_Angeles’	Returns a timestamp with time zone data type even without specifying timestamp with time zone format. '1947-08-05 10:45:00.221111000 AM AMERICA/LOS_ANGELES'
‘1947-08-05 10:45:00.221111000 AM America/Los_Angeles’, ’MM-DD-YYYY HH:MI:SS.NS AM'	Default date time format at the Run Configurations dialog is used when the format is not specified at the function level. Default date time format: 'YYYY-MM-DD HH:MI:SS.NS AM TZR' If a timestamp with time zone data does not match the given format, the following error appears: Process row failed for function [TO_TIMESTAMP_TZ]: Failed to convert the string to timestamp with time zone value. Verify that the specified date format string is valid. Verify that the timestamp with time zone string used in the first argument is compatible with the specified date format.

TRUNC (Dates)

Truncates dates to a specific year, month, day, hour, minute, second, millisecond, or microsecond. You can also use TRUNC to truncate numbers.

You can truncate the following date parts:

- **Year.** If you truncate the year portion of the date, the function returns Jan 1 of the input year with the time set to 00:00:00.000000000. For example, the following expression returns 1/1/1997 00:00:00.000000000:

  ```
  TRUNC(12/1/1997 3:10:15, 'YY')
  ```

- **Month.** If you truncate the month portion of a date, the function returns the first day of the month with the time set to 00:00:00.000000000. For example, the following expression returns 4/1/1997 00:00:00.000000000:

  ```
  TRUNC(4/15/1997 12:15:00, 'MM')
  ```

- **Day.** If you truncate the day portion of a date, the function returns the date with the time set to 00:00:00.000000000. For example, the following expression returns 6/13/1997 00:00:00.000000000:

  ```
  TRUNC(6/13/1997 2:30:45, 'DD')
  ```

- **Hour.** If you truncate the hour portion of a date, the function returns the date with the minutes, seconds, and subseconds set to 0. For example, the following expression returns 4/1/1997 11:00:00.000000000:

  ```
  TRUNC(4/1/1997 11:29:35, 'HH')
  ```

- **Minute.** If you truncate the minute portion of a date, the function returns the date with the seconds and subseconds set to 0. For example, the following expression returns 5/22/1997 10:15:00.000000000:

  ```
  TRUNC(5/22/1997 10:15:29, 'MI')
  ```

- **Second.** If you truncate the second portion of a date, the function returns the date with the milliseconds set to 0. For example, the following expression returns 5/22/1997 10:15:29.000000000:

  ```
  TRUNC(5/22/1997 10:15:29.135, 'SS')
  ```

- **Millisecond.** If you truncate the millisecond portion of a date, the function returns the date with the microseconds set to 0. For example, the following expression returns 5/22/1997 10:15:30.135000000:

  ```
  TRUNC(5/22/1997 10:15:30.135235, 'MS')
  ```

- **Microsecond.** If you truncate the microsecond portion of a date, the function returns the date with the nanoseconds set to 0. For example, the following expression returns 5/22/1997 10:15:30.135235000:

  ```
  TRUNC(5/22/1997 10:15:29.135235478, 'US')
  ```

Syntax

```
TRUNC( date [,format] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
date	Required	Date/Time datatype. The date values you want to truncate. You can enter any valid transformation expression that evaluates to a date.
format	Optional	Enter a valid format string. The format string is not case sensitive. If you omit the format string, the function truncates the time portion of the date, setting it to 00:00:00.000000000.

Return Value

Date.

NULL if a value passed to the function is NULL.

Examples

The following expressions truncate the year portion of dates in the DATE_SHIPPED port:

```
TRUNC( DATE_SHIPPED, 'Y' )
TRUNC( DATE_SHIPPED, 'YY' )
TRUNC( DATE_SHIPPED, 'YYY' )
TRUNC( DATE_SHIPPED, 'YYYY' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 1 1998 00:00:00.000000000
Apr 19 1998 1:31:20PM	Jan 1 1998 00:00:00.000000000
Jun 20 1998 3:50:04AM	Jan 1 1998 00:00:00.000000000
Dec 20 1998 3:29:55PM	Jan 1 1998 00:00:00.000000000
NULL	NULL

The following expressions truncate the month portion of each date in the DATE_SHIPPED port:

```
TRUNC( DATE_SHIPPED, 'MM' )
TRUNC( DATE_SHIPPED, 'MON' )
TRUNC( DATE_SHIPPED, 'MONTH' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 1 1998 00:00:00.000000000
Apr 19 1998 1:31:20PM	Apr 1 1998 00:00:00.000000000
Jun 20 1998 3:50:04AM	Jun 1 1998 00:00:00.000000000
Dec 20 1998 3:29:55PM	Dec 1 1998 00:00:00.000000000
NULL	NULL

The following expressions truncate the day portion of each date in the DATE_SHIPPED port:

```
TRUNC( DATE_SHIPPED, 'D' )
TRUNC( DATE_SHIPPED, 'DD' )
TRUNC( DATE_SHIPPED, 'DDD' )
TRUNC( DATE_SHIPPED, 'DY' )
TRUNC( DATE_SHIPPED, 'DAY' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 15 1998 00:00:00.000000000
Apr 19 1998 1:31:20PM	Apr 19 1998 00:00:00.000000000
Jun 20 1998 3:50:04AM	Jun 20 1998 00:00:00.000000000
Dec 20 1998 3:29:55PM	Dec 20 1998 00:00:00.000000000
Dec 31 1998 11:59:59PM	Dec 31 1998 00:00:00.000000000
NULL	NULL

The following expressions truncate the hour portion of each date in the DATE_SHIPPED port:

```
TRUNC( DATE_SHIPPED, 'HH' )
TRUNC( DATE_SHIPPED, 'HH12' )
TRUNC( DATE_SHIPPED, 'HH24' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:31AM	Jan 15 1998 02:00:00.000000000
Apr 19 1998 1:31:20PM	Apr 19 1998 13:00:00.000000000
Jun 20 1998 3:50:04AM	Jun 20 1998 03:00:00.000000000
Dec 20 1998 3:29:55PM	Dec 20 1998 15:00:00.000000000
Dec 31 1998 11:59:59PM	Dec 31 1998 23:00:00.000000000
NULL	NULL

The following expression truncates the minute portion of each date in the DATE_SHIPPED port:

```
TRUNC( DATE_SHIPPED, 'MI' )
```

DATE_SHIPPED	RETURN VALUE
Jan 15 1998 2:10:30AM	Jan 15 1998 02:10:00.000000000
Apr 19 1998 1:31:20PM	Apr 19 1998 13:31:00.000000000
Jun 20 1998 3:50:04AM	Jun 20 1998 03:50:00.000000000
Dec 20 1998 3:29:55PM	Dec 20 1998 15:29:00.000000000
Dec 31 1998 11:59:59PM	Dec 31 1998 23:59:00.000000000
NULL	NULL

TRUNC (Numbers)

Truncates numbers to a specific digit. You can also use TRUNC to truncate dates.

Syntax

```
TRUNC( numeric_value [, precision] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values you want to truncate. You can enter any valid transformation expression that evaluates to a Numeric datatype.
precision	Optional	Can be a positive or negative integer. You can enter any valid transformation expression that evaluates to an integer. The integer specifies the number of digits to truncate.

If *precision* is a positive integer, TRUNC returns *numeric_value* with the number of decimal places specified by *precision*. If *precision* is a negative integer, TRUNC changes the specified digits to the left of the decimal point to zeros. If you omit the *precision* argument, TRUNC truncates the decimal portion of *numeric_value* and returns an integer.

If you pass a decimal *precision* value, the Data Integration Service rounds *numeric_value* to the nearest integer before evaluating the expression.

When you run a mapping in high precision mode, use the ROUND function before truncating.

For example, suppose the following expression is used to truncate the values in the QTY port:

```
TRUNC ( QTY / 15 )
```

When the value for QTY = 15000000, the session returns the value 999999. The expected result is 1000000.

At run time, the Data Integration Service evaluates the constant part of the expression and then the variable part.

In the above expression, QTY is the variable value and (1/15) is the constant value.

When QTY = 15000000, the expression is evaluated as follows:

```
TRUNC ( 15000000 * (1/15)
TRUNC ( 15000000 * (1/15)
= TRUNC ( 15000000 * 0.0666666666666666)
= TRUNC ( 15000000 * 0.0666666666666666)
= TRUNC ( 999999.99999999)
= 999999
```

If you use the ROUND function before truncating, the expression is evaluated as follows:

```
TRUNC (ROUND (QTY/15, .99999999999999999999999999)).
```

Return Value

Numeric value.

NULL if one of the arguments is NULL.

Note: If the return value is Decimal with precision greater than 15, you can enable high precision to ensure decimal precision up to 38 digits.

Examples

The following expressions truncate the values in the Price port:

```
TRUNC( PRICE, 3 )
```

PRICE	RETURN VALUE
12.9995	12.999
-18.8652	-18.865
56.9563	56.956
15.9928	15.992
NULL	NULL

```
TRUNC( PRICE, -1 )
```

PRICE	RETURN VALUE
12.99	10.0
-187.86	-180.0
56.95	50.0
1235.99	1230.0
NULL	NULL

```
TRUNC( PRICE )
```

PRICE	RETURN VALUE
12.99	12.0

PRICE	RETURN VALUE
-18.99	-18.0
56.95	56.0
15.99	15.0
NULL	NULL

UPPER

Converts lowercase string characters to uppercase.

Syntax

```
UPPER( string )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
string	Required	String datatype. Passes the values you want to change to uppercase text. You can enter any valid transformation expression.

Return Value

Uppercase string. If the data contains multibyte characters, the return value depends on the code page and data movement mode of the Data Integration Service.

NULL if a value passed to the function is NULL.

Example

The following expression changes all names in the FIRST_NAME port to uppercase:

```
UPPER( FIRST_NAME )
```

FIRST_NAME	RETURN VALUE
Ramona	RAMONA
NULL	NULL
THOMAS	THOMAS
PierRe	PIERRE
Bernice	BERNICE

UUID4

Returns a randomly generated 16-byte binary value that complies with variant 4 of the UUID specification described in RFC 4122. UUID4 does not take an argument.

Syntax

```
UUID4()
```

Return Value

Binary.

UUID4 never returns a null value or an error.

UUID_UNPARSE

Converts a 16-byte binary value to a 36-character string representation as specified in RFC 4122.

Syntax

```
UUID_UNPARSE( binary )
```

The following table describes the argument for this command:

Argument	Required/ Optional	Description
binary	Required	Binary datatype. Any 16-byte binary value that you want to convert to a 36-character string.

Return Value

36-character string.

Returns null if the argument is null and an error if the argument is not a 16-byte binary value.

Example

The following expression might return a value of 6948DF80-14BD-4E04-8842-7668D9C001F5:

```
UUID_UNPARSE(UUID4())
```

VARIANCE

Returns the variance of a value you pass to it. VARIANCE is used to analyze statistical data. You can nest only one other aggregate function within VARIANCE, and the nested function must return a Numeric datatype.

Syntax

```
VARIANCE( numeric_value [, filter_condition ] )
```

The following table describes the arguments for this command:

Argument	Required/ Optional	Description
numeric_value	Required	Numeric datatype. Passes the values for which you want to calculate a variance. You can enter any valid transformation expression.
filter_condition	Optional	Limits the rows in the search. The filter condition must be a numeric value or evaluate to TRUE, FALSE, or NULL. You can enter any valid transformation expression.

Return Value

Double value.

NULL if all values passed to the function are NULL or if no rows are selected (for example, the *filter_condition* evaluates to FALSE or NULL for all rows).

Nulls

If a single value is NULL, VARIANCE ignores it. However, if all values passed to the function are NULL or if no rows are selected, VARIANCE returns NULL.

Group By

VARIANCE groups values based on group by ports you define in the transformation, returning one result for each group.

If there is not a group by port, VARIANCE treats all rows as one group, returning one value.

Example

The following expression calculates the variance of all rows in the TOTAL_SALES port:

```
VARIANCE( TOTAL_SALES )
```

```
TOTAL_SALES
2198.0
2256.0
3001.0
NULL
8953.0
RETURN VALUE:  10592444.6666667
```

INDEX

A

ABORT function
  description 56
ABS function
  description 56
absolute values
  obtaining 56
ADD_TO_DATE function
  description 57
Advanced Encryption Standard algorithm
  description 60, 61
AES_DECRYPT function
  description 60
AES_ENCRYPT function
  description 61
aggregate functions
  ANY 62
  AVG 65
  COUNT 78
  description 48
  FIRST 93
  LAST 118
  MAX (dates) 134
  MAX (numbers) 135
  MAX (string) 136
  MEDIAN 138
  MIN (dates) 143
  MIN (numbers) 144, 146
  null values 19, 50
  PERCENTILE 154
  STDDEV 192
  SUM 198
  VARIANCE 226
AND
  reserved word 15
ANY function
  description 62
arithmetic
  date/time values 47
arithmetic operators
  description 30
  using strings in expressions 30
  using to convert data 30
array
  generating 63, 71, 132, 133
ARRAY function
  description 63
ASCII
  CHR function 69
  converting ASCII values 69
  converting characters to ASCII values 64
  converting to Unicode values 70
ASCII function
  description 64
averages
  aggregate functions for determining 65
  returning 148
AVG function
  description 65

B

bigint
  converting values to 203
built-in variables
  description 34

C

calendars
  date types supported 36
capitalization
  strings 106, 125, 225
case
  converting to uppercase 225
CAST function
  description 66
CEIL function
  description 67
character functions
  ASCII 64
  CHR 69
  CHRCODE 70
  CONCAT function 73
  INITCAP 106
  INSTR 107
  LENGTH 123
  list of 50
  LOWER 125
  LPAD 126
  LTRIM 128
  METAPHONE 140
  REG_EXTRACT 160
  REG_MATCH 163
  REG_REPLACE 164
  REPLACECHR 165
  REPLACESTR 168
  RPAD 179
  RTRIM 180
  SOUNDEX 188
  SUBSTR 196
  UPPER 225
character strings
  converting from dates 205
  converting to dates 211
characters
  adding to strings 126, 179
  ASCII characters 64, 69

characters *(continued)*
capitalization 106, 125, 225
counting 196
encoding 140, 188
removing from strings 128, 180
replacing multiple 168
replacing one 165
returning number 123
Unicode characters 64, 69, 70
CHOOSE function
description 68
CHR function
description 69
inserting single quotes 13, 69
CHRCODE function
description 70
COBOL syntax
converting to perl syntax 160
COLLECT_LIST function
description 71
COLLECT_MAP function
description 71
comments
adding to expressions 14
comparison operators
description 31
using strings in expressions 31
complex functions
ARRAY 63
CAST 66
COLLECT_LIST 71
COLLECT_MAP 71
CONCAT_ARRAY 75
description 51
EXTRACT_STRUCT 93
MAP 130
MAP_FROM_ARRAYS 131
MAP_KEYS 132
MAP_VALUES 133
PARSE_JSON function 152
PARSE_XML function 153
RESPEC 171
SIZE 187
STRUCT 193
STRUCT_AS 195
complex operators
accessing nested data types 26
description 22
for array with struct elements 27
for multidimensional arrays 26
for nested data types 26
for struct with array elements 29
for struct with struct elements 29
using to access data 22
COMPRESS function
description 72
compression
compressing data 72
decompressing data 90
CONCAT function
description 73
inserting single quotes using 73
CONCAT_ARRAY function
description 75
concatenating
strings 31, 73
constants
DD_INSERT 16
constants *(continued)*
DD_REJECT 17
DD_UPDATE 17
description 11
FALSE 18
NULL 18
TRUE 19
conversion functions
CREATE_TIMESTAMP_TZ 81
description 52
GET_TIMESTAMP 99
GET_TIMEZONE 99
TO_CHAR (dates) 205
TO_CHAR (numbers) 209
TO_DATE 211
TO_DECIMAL 214
TO_DECIMAL38 215
TO_FLOAT 216
TO_INTEGER 217
TO_TIMESTAMP_TZ 219
CONVERT_BASE function
description 75
converting
date strings 37
COS function
description 76
COSH function
description 77
cosine
calculating 76
calculating hyperbolic cosine 77
COUNT function
description 78
CRC32 function
description 80
CREATE_TIMESTAMP_TZ function
description 81
CUME function
description 82

D

data cleansing functions
description 52
GREATEST 100
IN 104
LEAST 122
Data Integration Service
handling nulls in comparison expressions 19
datatypes
Date/Time 35
date functions
ADD_TO_DATE 57
DATE_COMPARE 83
DATE_DIFF 84
GET_DATE_PART 97
LAST_DAY 119
MAKE_DATE_TIME 129
MAX (dates) 134
MIN (dates) 143
ROUND 173
SET_DATE_PART 182
SYSTIMESTAMP 199
TRUNC (Dates) 220
DATE_COMPARE function
description 83

DATE_DIFF function
description 84
date/time values
adding 57
dates
converting to character strings 205
default datetime format 38
flat files 38
format strings 39
functions 53
Julian 36
Modified Julian 36
overview 35
performing arithmetic 47
relational databases 38
rounding 173
truncating 220
year 2000 36
DD_DELETE constant
description 16
reserved word 15
update strategy example 16
DD_INSERT constant
description 16
reserved word 15
update strategy example 16
DD_REJECT constant
description 17
reserved word 15
update strategy example 17
DD_UPDATE constant
description 17
reserved word 15
update strategy example 17
DEC_BASE64 function
description 87
decimal values
converting 81, 99, 214, 215, 219
DECODE function
description 88
internationalization 12
decoding
DEC_BASE64 function 87
DECOMPRESS function
description 90
decryption
AES_DECRYPT function 60
default datetime format
setting 38
default values
ERROR function 91
division calculation
returning remainder 147
dot operator
description 24
for complex data types 22
using to access data 24
dot operators
for nested data type 26
for struct with struct elements 29
double precision values
floating point numbers 216

E

empty strings
testing for 123
ENC_BASE64 function
description 90
encoding
characters 140, 188
ENC_BASE64 function 90
encoding functions
AES_DECRYPT 60
AES_ENCRYPT 61
COMPRESS 72
CRC32 80
DEC_BASE64 87
DECOMPRESS 90
description 53
ENC_BASE64 90
MD5 137
encryption
AES_ENCRYPT function 61
using the Advanced Encryption Standard algorithm 61
ERROR function
default value 91
description 91
EXP function
description 92
exponent values
calculating 92
returning 157
expressions
adding comments 14
conditional 18
overview 11
syntax 12
using operators 21
EXTRACT_STRUCT function
description 93

F

FALSE constant
description 18
reserved word 15
filter conditions
aggregate functions 50
null values 19
Filter transformation
using ISNULL function 110
financial functions
description 54
FV function 96
NPER function 151
PMT function 156
PV function 158
RATE function 160
FIRST function
description 93
flat files
dates 38
FLOOR function
description 95
FLOOR function (expressions)
description 95
format
from character string to date 211
from date to character string 205
format strings
dates 39
definition 35
IS_DATE function 43

format strings *(continued)*
Julian day 40, 43
matching 45
Modified Julian day 40, 43
TO_CHAR function 40
TO_DATE function 43
functions
aggregate 48
categories 48
character 50
complex 51
conversion 52
data cleansing 52
date 53
description 11
encoding 53
financial 54
internationalization 12
numeric 54
scientific 54
special 55
string 55
test 55
window 55
FV function
description 96

G

GET_DATE_PART function
description 97
GET_TIMESTAMP function
description 99
GET_TIMEZONE function
description 99
GREATEST function
description 100
Gregorian calendar
in date functions 36

H

hierarchical data
accessing elements 22
generating 63, 71, 93, 130–133, 152, 153, 193, 195
parsing 152, 153
high precision
ABS 56
ABS function 56
arithmetic operators 30
AVG 65
AVG function 65
CEIL 67
CREATE_TIMESTAMP_TZ function 81
CUME 82
CUME function 82
EXP 92
GET_TIMESTAMP function 99
GET_TIMEZONE function 99
LOG 125
MAX (numbers) 135
MAX function 135
MEDIAN 138
MEDIAN function 138
MIN (numbers) 144
MIN function 144
high precision *(continued)*
MOD 147
MOVINGAVG 148
MOVINGAVG function 148
MOVINGSUM 150
MOVINGSUM function 150
PERCENTILE 154
PERCENTILE function 154
POWER 157
ROUND (numbers) 177
ROUND function 177
SIGN 184
SIN 185
STDDEV function 192
SUM 198
SUM function 198
TO_DECIMAL function 214
TO_DECIMAL38 function 215
TO_TIMESTAMP_TZ function 219
TRUNC function 223
hyperbolic
cosine function 77
sine function 186
tangent function 201

I

IIF function
description 101
internationalization 12
IN function
description 104
INDEXOF function
description 105
:INFA reference qualifier
reserved word 15
INITCAP function
description 106
internationalization 12
INSTR function
description 107
integers
converting values to 217
internationalization
functions affected 12
invalid expression 12
sort order 12
IS_DATE function
description 111
format strings 43
IS_NUMBER function
description 113
IS_SPACES function
description 115
ISNULL function
description 110

J

J format string
using with IS_DATE 46
using with TO_CHAR 42
using with TO_DATE 46
Julian dates
in date functions 36

Julian day
format string 40, 43

L

:LKP reference qualifier
description 13
reserved word 15
LAG function
description 116
LAST function
description 118
LAST_DAY function
description 119
LEAD function
description 120
LEAST function
description 122
LENGTH function
description 123
empty string test 123
literals
single quotes in 69, 73
single quotes requirement 13
LN function
description 124
local variables
description 11
LOG function
description 125
logarithm
returning 124, 125
logical operators
description 33
LOWER function
description 125
internationalization 12
LPAD function
description 126
LTRIM function
description 128

M

MAKE_DATE_TIME function
description 129
map
generating 71, 130, 131
MAP function
description 130
MAP_FROM_ARRAYS function
description 131
MAP_KEYS function
description 132
MAP_VALUES function
description 133
mapping parameters
definition 11
mapping variables
built-in variables 34
MAX (dates) function
description 134
internationalization 12
MAX (numbers) function
description 135
internationalization 12
MAX (string) function
description 136
:MCR reference qualifier
reserved word 15
MD5 function
description 137
MEDIAN function
description 138
METAPHONE
description 140
MIN (dates) function
description 143
internationalization 12
MIN (numbers) function
description 144, 146
internationalization 12
minimum
value, returning 143
MOD function
description 147
Modified Julian day
format string 40, 43
month
returning last day 119
MOVINGAVG function
description 148
MOVINGSUM function
description 150
multiple searches
example of TRUE constant 20

N

negative values
SIGN 184
nested expressions
operators 21
NOT
reserved word 15
NPER function
description 151
NULL constant
description 18
reserved word 15
null values
aggregate functions 19, 50
checking for 110
filter conditions 19
in comparison expressions 19
ISNULL 110
logical operators 33
operators 19
string operator 31
numbers
rounding 177
truncating 223
numeric functions
ABS 56
CEIL 67
CONVERT_BASE 75
CUME 82
description 54
EXP 92
FLOOR 95
LN 124
LOG 125
MOD 147

numeric functions *(continued)*
MOVINGAVG 148
MOVINGSUM 150
POWER 157
RAND 159
ROUND (numbers) 177
SIGN 184
SQRT 191
TRUNC (numbers) 223
numeric values
converting to text strings 209
returning absolute value 56
returning cosine 76
returning hyperbolic cosine of 77
returning hyperbolic sine 186
returning hyperbolic tangent 201
returning logarithms 124, 125
returning minimum 144
returning sine 185
returning square root 191
returning standard deviation 192
returning tangent 200
SIGN 184

O

operator precedence
expressions 21
operators
arithmetic 30
comparison operators 31
complex 22
description 11
logical operators 33
null values 19
string operators 31
using strings in arithmetic 30
using strings in comparison 31
OR
reserved word 15

P

PARSE_JSON function
description 152
PARSE_XML function
description 153
PERCENTILE function
description 154
perl compatible regular expression syntax
using in a REG_EXTRACT function 160
using in a REG_MATCH function 160
PMT function
description 156
ports
syntax 13
positive values
SIGN 184
POWER function
description 157
primary key constraint
null values 18
PROC_RESULT variable
reserved word 15
PV function
description 158

Q

quotation marks
inserting single using CHR function 13

R

RAND function
description 159
RATE function
description 160
reference qualifiers
description 13
REG_EXTRACT function
description 160
using perl syntax 160
REG_MATCH function
description 163
using perl syntax 160
REG_REPLACE function
description 164
relational databases
dates 38
REPLACECHR function
description 165
REPLACESTR function
description 168
reserved words
list 15
RESPEC function
description 171
return values
description 11
syntax 13
REVERSE function
description 172
ROUND (dates) function
description 173
processing subseconds 173
ROUND (numbers) function
description 177
rounding
dates 173
numbers 177
rows
avoiding spaces 115
counting 78
returning any row 62
returning average 148
returning first row 93
returning last row 118
returning sum 150
running total 82
skipping 91
RPAD function
description 179
RR format string
description 37
difference between YY and RR 37
using with IS_DATE 46
using with TO_CHAR 43
using with TO_DATE 46
RTRIM function
description 180
running total
returning 82

S

scientific functions
  COS 76
  COSH 77
  description 54
  SIN 185
  SINH 186
  TAN 200
  TANH 201
SESSSTARTTIME variable
  using in date functions 47
SET_DATE_PART function
  description 182
SIGN function
  description 184
SIN function
  description 185
sine
  returning 185, 186
single quotes in string literals
  CHR function 69
  using CHR and CONCAT functions 73
SINH function
  description 186
size
  array 187
  map 187
SIZE function
  description 187
skipping
  rows 91
sort order
  internationalization 12
SOUNDEX function
  description 188
spaces
  avoiding in rows 115
  removing with DD_REJECT 17
special functions
  ABORT 56
  DECODE 88
  description 55
  ERROR 91
  IIF 101
SPOUTPUT
  reserved word 15
SQL IS_CHAR function
  using REG_MATCH 163
SQL LIKE function
  using REG_MATCH 163
SQL syntax
  converting to perl syntax 160
SQL_LIKE function
  description 190
SQRT function
  description 191
square root
  returning 191
SSSSS format string
  using with IS_DATE 46
  using with TO_CHAR 42
  using with TO_DATE 46
standard deviation
  returning 192
STDDEV function
  description 192
string conversion
  dates 37
string functions
  CHOOSE 68
  description 55
  INDEXOF 105
  REVERSE 172
string literals
  single quotes in 69, 73
  single quotes requirement 13
string operators
  description 31
string values
  returning maximum 136
  returning minimum 146
strings
  adding blanks 126
  adding characters 126
  capitalization 106, 125, 225
  character set 107
  concatenating 31, 73
  converting character strings to dates 211
  converting dates to characters 205
  converting length 179
  converting numeric values to text strings 209
  number of characters 123
  removing blanks 128
  removing blanks and characters 180
  removing characters 128
  replacing multiple characters 168
  replacing one character 165
  returning portion 196
struct
  generating 93, 152, 153, 193, 195
STRUCT function
  description 193
STRUCT_AS function
  description 195
subscript operator
  array data type 23
  for complex data types 22
  map data type 23
  using to access data 23
subscript operators
  for multidimensional arrays 26
  for nested data type 26
subseconds
  processing in ROUND (dates) function 173
  processing in TRUNC (dates) function 220
SUBSTR function
  description 196
sum
  returning 150, 198
SUM function
  description 198
syntax
  expression 12
  general rules 13
  ports 13
  return values 13
SYSDATE variable
  description 34
  reserved word 15
  using in expressions 34
system variables 34
SYSTIMESTAMP function
  description 199

T

TAN function
  description 200
tangent
  returning 200, 201
TANH function
  description 201
test functions
  description 55
  IS_DATE 111
  IS_NUMBER 113
  IS_SPACES 115
  ISNULL 110
text strings
  converting numeric values 209
TO__TIMESTAMP_TZ function
  description 219
TO_CHAR (dates) function
  description 205
  examples 42
  format strings 40
TO_CHAR (numbers) function
  description 209
TO_DATE function
  description 211
  examples 45
  format strings 43
TO_DECIMAL function
  description 214
TO_DECIMAL38 function
  description 215
TO_FLOAT function
  description 216
TO_INTEGER function
  description 217
transformation expressions
  null constraints 18
  overview 11
transformation language
  compared to SQL 12
  operators 21
  reserved words 15
transformation language components
  overview 11
transformation language updates
  boolean expressions 19
  comparison expressions 19
TRUE constant
  description 19
  reserved word 15
TRUNC (dates) function
  description 220
  processing subseconds 220
TRUNC (numbers) function
  description 223
truncating
  dates 220
  numbers 223
:TYPE reference qualifier
  reserved word 15

U

Unicode
  converting characters to Unicode values 64
  converting to ASCII values 70
  converting Unicode values 69
update strategy
  DD_DELETE example 16
  DD_INSERT example 16
  DD_REJECT example 17
  DD_UPDATE example 17
UPPER function
  description 225
  internationalization 12
UUID_UNPARSE function
  description 226
UUID4 function
  description 226

V

variables
  built-in variables 34
  SYSDATE 34
VARIANCE function
  description 226

W

window functions
  description 55
  LAG 116
  LEAD 120

Y

year 2000
  dates 36
YY format string
  difference between RR and YY 37
  using with IS_DATE 46
  using with TO_CHAR 43
  using with TO_DATE 46