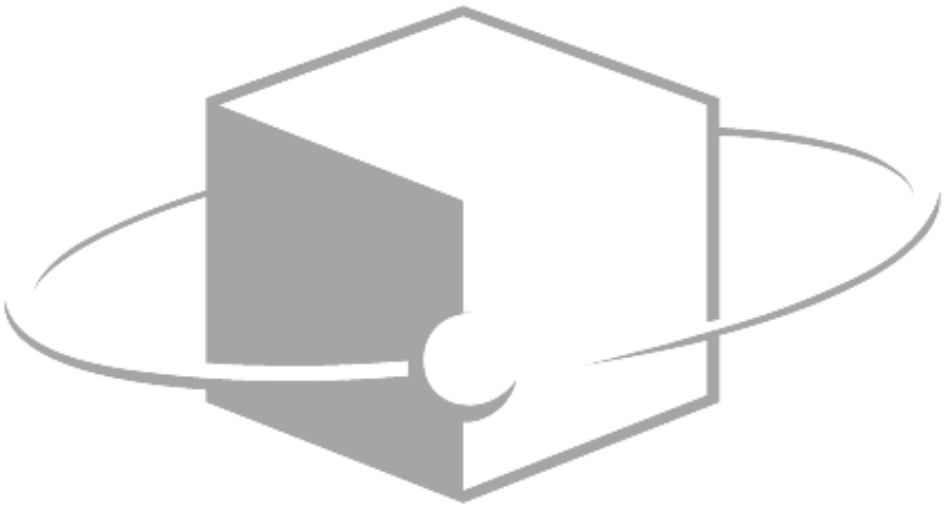
instantOLAP 3.1



FUNCTION REFERENCE

Internal expression language and functions of instantOLAP

Version 3.1, 5/7/2013

Before you start using this guide, it is important to understand the terms and typographical conventions used in the documentation. The following kinds of formatting in the text identify special information.

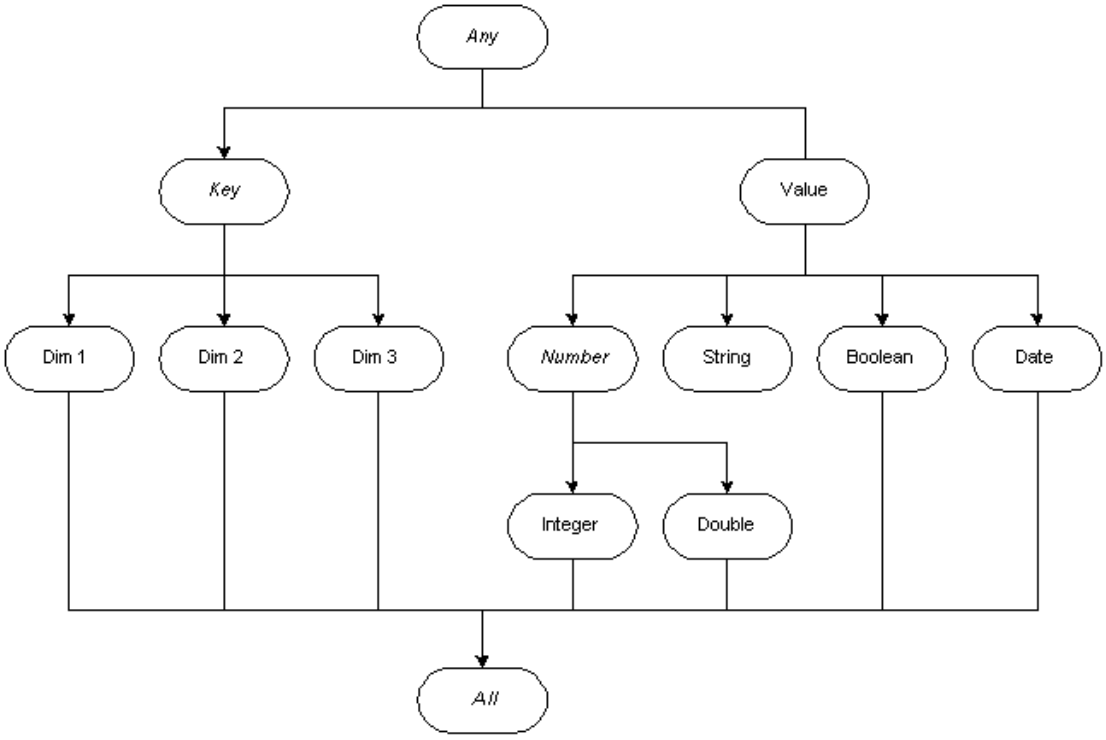
Formatting convention	**Type of Information**
Special Bold	Items you must select, such as menu options, command buttons, or items in a list.
Brackets (<>)	Used for variable expressions such as parameters
`Monospace font`	Marks code examples or an expression-syntax
Emphasis	Use to emphasize the importance of a point
CAPITALS	Names of keys on the keyboard. for example, SHIFT, CTRL, or ALT
KEY+KEY	Key combinations for which the user must press and hold down one key and then press another, for example, CTRL+P, or ALT+F4

Content

The expression language

Type-system

instantOLAP uses a type system (known from programming languages), where all values belong to a specific type. With the type system, instantOLAP can perform syntax checks and generate error messages for your expressions.



The type-system is organized in a hierarchy, with types, subtypes and super-types. Some types are instanceable, which means values with exactly this type can exist. Other types are only super-types, which group other types but can never have instances. "Value" is such a super-type.

A super-type of a type is another type, which lies above the type in the hierarchy. For instance, "Value" is a super-type of "Integer". A subtype is a type which lies under a type in the hierarchy. If a function expects an argument of type X you can pass any value, which has exactly this type X or a subtype of X. I.e. a function expecting an argument of the type "Number" would accept both values of the type "Integer" or "Double".

Note that dimensions automatically become types in the graph and are always a subtype of "Key". Because of this, each function which expects arguments of the type "Key" will accepts keys from every dimension.

Any

The type "Any" is the basic type in the type-system - every other type is a subtype of "Any" and every value belongs to the type "Any". If a function expects an argument of the type "Any", you can pass anything you want.

There can be no instance of the type "Any" itself.

Key

All elements (keys) of a dimension have the type "Key" (more exact: each dimension becomes its own type in the model, which is a subtype of "Key". The elements of the dimension have their dimension as type). Keys are no value so the only super type of "Key" is "Any".

Value

The type "Value" is the super type for all kinds of values - Strings, Numbers, Booleans, Dates and so on. No value has really the type "Value", because every value belongs to a subtype of this.

Number

"Number" is another type, which groups all kinds of numbers (Integers and Doubles). No number belongs to this type directly because each number is an instance of a subtype of it.

Double

The type "Double" represents numbers with a fraction. The type "Double" is a subtype of "Number".

Integer

Integers are numbers without any fractions. The type "Integer" is a subtype of "Number".

String

Strings have the type "String", which is a subtype of "Value".

Boolean

Boolean-values are logical values, which can have the values "true" or "false". The type Boolean is a subtype of "Value".

Date

The type "Date" represents a time stamp (including years, months, days, hours, minutes, seconds and milliseconds).

All

The special type "All" is a sub-type of all other types. The only value having "All" as type is NULL, which means NULL can be passed for any argument because NULL fits to every type.

NULL always represents an "undefined" value.

Syntax

Dimensions and selections

Syntax

```
<dimension-syntax> := <dimension-name>
```

Result-Type

Key

Description

With the dimension-name itself, you can access a dimension respective the selection (the filter) of a dimension. In the filter, no, one or more keys are stored for each dimension. With the dimension-name, you can access these keys.

The filter is influenced by a number of query-elements, for instance by

- the URL of a query,
- the selectors,
- the filter of the query, the blocks, the pivot-tables or the headers and

- the iterations of blocks and headers.

Examples

```
Time
```

Returns the current selection of the dimension "Time"

Dimension levels

Syntax

```
<level-expression> := <dimension-name> '::' <level-name>
```

```
<level-expression> := <level-name>
```

Result-Type

Key

Description

Similar to the access to dimensions and selections, you can access the filtered keys of a specific level of a dimension. However, in difference to the dimension-access, this will not return the selected keys directly but all keys above or underneath the selected keys, depending on the desired level and on its position (above or underneath the level of the level of the selected keys).

Examples

If the currently selected key of the dimension "Time" was "Feb/2011", the expression

```
Time::Year
```

would result to Time:'2011' and

```
Time::Day
```

to Time:'01.02.2011' + Time:'02.02.2011' + ... + Time:'29.02.2011'

Operators

Using operators

Some functions are also available as operators which you can use instead of writing down the whole function-name. A few operators have no corresponding function.

Operators have a priority which defines the order in which the engine will evaluate them. The engine will evaluate the operators starting with the highest priority, down to the smallest. If two operators have the same priority, they will be evaluated from the left to the right.

If you want to change the execution-order of your expression, you must use brackets to override the operators' priorities.

List of operators

Operator	Priority	Function	Example
:	6	Dimension keys	`Time:'2011'`
::	6	Dimension levels	`Time::YEAR`
$	6	Accessing variables	`$QUERYNAME`
.	5	Accessing attributes	`Time.week`
[]	5	CUBE	`[CHILDREN(Product)]`
{}	5	TOSTRING	`{Amount()}`
*	4	MUL	`10 * 42`
/	4	DIV	`10 / 42`
%	4	MOD	`10 % 42`
+	3	ADD	`10 + 20`
-	3	SUB	`20 - 10`
|	3	JOIN	`10 | 20 | 30`
IN	3	IN	`Product IN Manufaturer.Products`
<	2	LESS	`10 < 20`
>	2	GREATER	`10 > 20`
=	2	EQUAL	`10 = 20`
<=	2	LESS_OR_EQUAL	`10 <= 20`
>=	2	GREATER_OR_EQUAL	`10 >= 20`
<>	2	UNEQUAL	`10 <> 20`
LIKE	2	LIKE	`Hello world' LIKE '*world'`
?	2	MATCH	`PRODUCT ? Product.color =`

Operator	Priority	Function	Example
			`'red'`
!	2	NOT	`!FALSE`
AND	1	AND	`Product.color = 'red' AND Amount() > 0`
OR	1	OR	`Product.color = 'red' OR Amount() > 0`

Brackets

Syntax

```
<bracket-expression> := '(' <expression> ')'
```

Description

With brackets, you can change the execution-order of your expressions. Normally, expressions are executed in the order of the used operators' priorities, beginning with the highest. Operators with equal priorities are executed from the left to right. By encapsulating parts of your expression in brackets, you will force the system to evaluate the content of the brackets first before using their result with the outer operators. You also may encapsulate multiple brackets - then the expressions will be executed from the inner to the outer.

Note that functions also work like brackets: All arguments of a function-call will be evaluated first (beginning with the first argument), then the function will be executed and after all the result is used together with the remaining part of your expression. If you encapsulate multiple functions (using functions as arguments for other functions), the inner functions will be executed before the outer.

Brackets may be used for any type of expressions (booleans, keys, numbers, etc.).

Examples

```
( 10 + 20 ) * 30
```

= 900

```
10 + 20 * 30
```

= 610

Accessing attributes

Syntax

```
<attribute-expression> := <key-expression> '.' <attribute-name>
```

Return-Type

Equal to the attribute-type

Description

Each element (key) of a dimension may have no, one or more attributes. Attributes can be simple values (Strings, Numbers, Booleans etc.) or elements (Keys) from other dimensions. When attributes have the type "Key", this connection between two dimensions is called a "Link".

To access an attribute you must user the dot-operators '.' immediately after a key-expression (attributes can only be read from keys). The return-type of the operator is equal to the type of the attribute. E.g. if an attribute has the type "String", the return type of the attribute-expression is also "String". If the type of the attribute is "Key", the return-type is "Key" and you may use the attribute-operator again to access other attributes of the attribute.

If the previous key-expression results to NULL, the result of the attribute-expression is also NULL.

Examples

```
Product.Color
```

Color of the current product

```
Product.Vendor.Name
```

Name of the vendor of the current product

Accessing variables

Syntax

```
<variable-expression> := '$' <variable-name>
```

Return-Type

String

Description

instantOLAP allows the definition and usage of variables inside queries. Variables can be defined in two different ways:

- By adding selectors to your query, which let the user choose the value of a variable,
- by passing arguments to the server in the URL.

Variables defined with selectors

Each selector automatically creates a variable with the name of the selector. A selector influences the filter of a query, because the user can select one or more elements of a dimension, but they also create a variable with the name of the dimension (for instance, instance a selector for "Time" creates a variable named "Time").

If a selector does not influence a dimension (because you gave a named to which matches no dimension), only the variable will be created (for instance. A selector with the name "X" will create a variable named "X" but not influence any dimension if there is no dimension with this name).

Variables passed in the URL

Whenever a query is called from outside the system (e.g. with a constant link from another website) you can append additional parameters to the queries' URL. All parameters will be converted into variables.

Types and evaluation

Variables are always strings. If you want variables to be interpreted as different types, for example, as a number, you can use the EVAL function. The EVAL function expects a string as arguments, compiles it to a valid expression (or throws an error if the parsing fails) and returns the result of the dynamically created expression. This also lets you use variables in a very flexible way, i.e. you could offer a list of key-expressions inside your query and use them inside headers for the iteration.

Lists

Variables can have no, one or more values, depending on the way you defined them. If a variable was defined by a Single-Selector, it will select not more than one value. If it was defined by a Multiple-Selector, it can have more selections. When the variable was passed as an URL-appendix, each occurrence of the variable in the URL will become one value.

Predefined variables

- **$COUNTRY:** The COUNTRY variable contains the country-code of the current user. This is equivalent to the country-code configured in the browser.
- **$LANGUAGE:** The LANGUAGE variable contains the language-code of the current user. This is equivalent to the language-code configured in the browser.
- **$QUERYAUTHOR:** The author of the current query.
- **$QUERYFILENAME:** The filename of the query.
- **$QUERYFOLDER:** The path of the folder where the query is located.
- **$QUERYDATE:** The creation-date of the query.
- **$QUERYNAME:** The name of the query.
- **$QUERYPATH:** The full path of the query inside the repository.
- **$USER**: The USER variable contains the account name of the current user.

Examples

```
$COUNTRY
```

E.g. returns 'US' (if the current user has configured 'US' as his country in his browser)

```
$LANGUAGE
```

E.g. return 'de' (if the current user has configured 'de' as his preferred language in his browser)

```
$USER
```

E.G. returns 'admin' (if the current user is named 'admin')

```
EVAL( $X )
```

Evaluates the variable X interpreted as an expression

Function calls

Description

Like in programming-languages, instantOLAP supports functions for evaluating values at runtime. Each function has a name, certain expected arguments and a return type. Whenever you use a function inside an expression, the function-call will be interpreted as an expression of the return-type and can be passed to other functions as arguments (if they expect an argument of this type or a super-type of it).

To call a function you'll have to type the function-name first, followed by brackets surrounding the arguments which are separated by commas. The arguments itself can also be expressions, for instance, you can pass constants, functions or dimensions as arguments.

Examples

```
NEXT( Time )
```

```
FIRST( LEVEL( Time, 1 ) )
```

Level functions

Syntax

```
<levelfunction-expression> := <level-name> '(' [ <Key> { ',' <Key> } ] ')'
```

Description

For each level of each dimension, the system automatically creates a function which you can use to find the keys of levels underneath or above the currently selected keys, depending on the position of the level and the level of the selected keys. With these functions, you can form expressions

like "give me all months of the selected year" or "give me all product-groups of the selected products".

Additionally, you can pass keys as arguments to change the current selection (the filters) for this functions-calls.

Examples

```
Month()

Month( NEXT( Time ) )
```

Fact functions

Syntax

```
<fact-expression> := <fact-name> '(' [ <Key> { ',' <Key> } ')'
```

Description

For each fact, instantOLAP automatically generates a function which you can use to access the values of this fact from the cubes. Like in the CUBE-function you also can pass keys as arguments to this function, changing the current filter which is used to access the cubes. Read the documentation of the CUBE-function for a more detailed description.

Examples

```
Amount()

Amount( NEXT( Time ) )
```

Comments

Description

You can also add comments to expressions. To start a comment you must place a double-slash (//) to your expression, the parser will then skip the rest of the line and continue parsing from the beginning of the following line.

Examples

```
IIF( Product.color = 'red',
   'Product is red'       // red product
   'Product is not red'   // other color
)
```

Constants

For the most type you can use constants inside expressions:

- NULL as constant for the type “All”.
- TRUE and FALSE as constants for the type “Boolean”.
- Strings as constants for the type “String”.
- Integer numbers as constants for the type “Integer”.
- Double numbers as constants for the type “Double”.
- Dimension-Keys as constants for the type “Key”.

Boolean constants

Syntax

```
<Boolean-Constant> := 'TRUE' | 'true' | 'FALSE' | 'false'
```

Description

TRUE and FALSE (written in capital or small letters) are the only possible constants for the type Boolean.

Examples

```
IIF( TRUE, 10, 20 )
```

= 10

Integer constants

Syntax

```
<Integer-Constant> = ['-'] { <digit> }
```

Description

Integer-Constant represent frictionless numbers and have the type Integer.

Examples

```
0
-1
100
```

Double constants

Syntax

```
<Double-Constant> = ['-'] { <digit> } '.' { <digit> }
```

Description

Double-Constant represent numbers with fraction and have the type Double.

Examples

```
0.0

100.10

-10.100
```

String constants

Syntax

```
<String-Constant> := "'" { <character> } "'" | '"' {
<character> } '"'
```

Description

Strings are constants of the type String. Strings inside expressions must be embedded in delimiters (like in programming languages). You may use two different types of delimiters, either the char " or the char '. Delimiters can be embedded (the delimiter ' can be embedded in " and vice versa).

Examples

```
'Hello World'

"Jim's car"
```

Dimension keys

Syntax

```
<Key-Constant> = <dimension-name> ':' <id> | <dimension-name>
":'" <id> "'"
```

Description

Key-Constants point to a single K of a specific Dimension and have the type Key. Whenever you use Key-Constants in queries, the expression is not influenced by the current filter, because this is a constant link to a key.

Each Key-constants consists of the dimension-name followed by the char ":" and the ID of the reference key. If the ID contains white spaces or special chars, you must embed it within the delimiters " or '.

Examples

```
Product:Coffee

Time:'01.01.2004'
```

NULL

Syntax

```
<NULL-Constant> := 'NULL'
```

Description

Like in SQL databases, NULL is the constant representing an undefined value for all types. NULL is the only value having the type All (which is a subtype of all other types), so you might it at any position and for any argument of a function.

Understanding the behavior of NULL and of functions with NULL-arguments is very important for database- and OLAP-systems. I.e. if your reports query data out of a database, the database will return NULL for the value not being stored inside their tables (for example, when no turnaround was stored for a specific date). The CUBE-function then will also return NULL for this values and all functions working with this value will have to deal with this NULL-value.

The most functions also return NULL if any of their parameters is NULL because there is no possible calculation for a valid return-value. But some functions exist especially for the treatment of NULL-values, the most important functions are ISNULL, EXISTS, and ZERO.

Examples

```
Amount( Time:Dec/2004 )
```

Results to NULL if there is no amount stored for December 2004

```
DIV( NULL, 5 )
```

Always returns NULL (impossible to calculate)

Functions

Overview

Math

Boolean

Compare

Flow control

Text manipulation

Dimension information

Searching, filtering and sorting

Hierarchies

Filter control

List functions

Date and Time

GETYEAR NOW SYSDATE
TIMESTAMP TODATE

Locale

COUNTRY LANGUAGE LOCALE

User related

HASROLES HASUSER USER

Values and cubes

COALESCE CUBE ISNULL
LOOKUP ROWNUM SUM
TOINTEGER TONUMBER ZERO

Conversion

TODATE TOINTEGER TOKEY
TONUMBER TOSTRING

Matrix

COLSPAN DRILLKEY DRILLLEVEL
ITERATIONKEY IVALUE MATRIX
MAX_X MAX_Y MIN_X
MIN_Y ROWNUM ROWSPAN
SCREENY SUBTOTAL UNIT
X XHEADER Y
YHEADER

Analysis

ABC AVGKEY FORECAST
MAXKEY MINKEY YTD
YTDR

Runtime

EVAL ERROR

Debugging

CATCH DEBUG ERROR
RETURNTYPE TYPE

Function details

ABC

Syntax

```
<abc-expression> := 'ABC(' keys: <Key> ',' expression:
<Number> ',' upper-border: <Number> ',' lower-border: <Number>
')'
```

Since

2.1

Return-type

Key

Description

The ABC functions helps building an ABC analysis. Usually a ABC analysis is used to find the elements building the top x% or bottom x% of a specific fact, for instance, all products which are responsible for the top 10% turnover of a company.

With this function, you can analyze keys by any fact and find the keys with the (summarized) N% of the fact:

1. The first argument determines the list of keys you want to analyze.
2. The second argument is the formula you want to apply on the keys (usually a simple fact).
3. The third and fourth argument defines the boundaries (maximum as first and minimum as second percentage, expected as values between 0.0 and 1.0) of the group you want to extract from the keys.

If any of the arguments is NULL, the function will return NULL.

Examples

```
ABC( LEVEL( Products, 1 ), Amount(), 1.0, 0.5 )
```

Find the Products with the top 50% amount in total.

```
ABC( LEVEL( Products, 1 ), Amount(), 0.5, 0.0 )
```

Find the Products which only build the bottom 50% amount in total

ABS

Syntax

```
<abs-expression> := 'ABS(' value: <Number> ')'
```

Since

2.0

Return-type

Number

Description

Returns the absolute value of the argument:

- If the argument is not negative, the argument is returned.
- If the argument is negative, the result is the negation of the argument.
- If the argument is NULL, the function returns NULL.

Examples

```
ABS( 1.0 )
```

= 1.0

```
ABS( -1 )
```

= 1

```
ABS( 0 )
```

= 0

```
ABS( NULL )
```

= NULL

ADD

Syntax

```
<add-expression> := 'ADD(' values: <Any> [ ',' { values:
<Any>} ] ')'
```

Since

1.0

Return-type

Depends on the argument's type, see below.

Description

The ADD-function adds or concatenates values of any type. The behavior and return-type of this function depend on the argument's type:

- If all arguments are Integers, the return-value is the sum of all values (the return type is Integer)
- If all arguments are Numbers, the return-value is the sum of all values (the return type is Double)
- If all arguments are Keys, the return value is the joined list of all Keys (the return type is Key)
- If all arguments are Strings, the return value is the concatenated string of all arguments (the return type is String).
- Otherwise it concatenates all values converted to String.

The function ignores NULL-values, even when arguments contain NULLs, the function will return the sum or concatenation of all other values.

This function is not type-safe because it adds anything and does no type-checks. Use the SUM-function for real number-addition (the SUM-function only accepts and returns values of the type Number).

Instead of the ADD-function you also can use the operator "+".

Examples

```
ADD( 10, 20 )
```

= 30

```
ADD( 'Hello', ' ', 'World' )
```

= 'Hello World'

```
ADD( 'Hello', 10 )
```

= 'Hello10'

```
ADD( Product:A, Product:B )
```

= Product:A | Product:B

See also

SUM

ALL

Syntax

```
<all-expression> := 'ALL(' dimension/level: <Key> ')'
```

Since

2.1

Return-type

Key

Description

The ALL function returns all keys of a dimension or level:

- If the argument is a dimension name, it returns all keys if that dimension in their natural order
- If the argument is a level name, it returns all keys of that level
- Otherwise the function just returns the argument

Examples

```
ALL( Time )
```

Returns all time keys

```
ALL( WEEK )
```

Returns all weeks (one specific level of the dimension Time)

See also

FAMILY, LEVEL

ANCESTORS

Syntax

```
<ancestors-expression> := 'ANCESTORS(' keys: <Key> ')'
```

Since

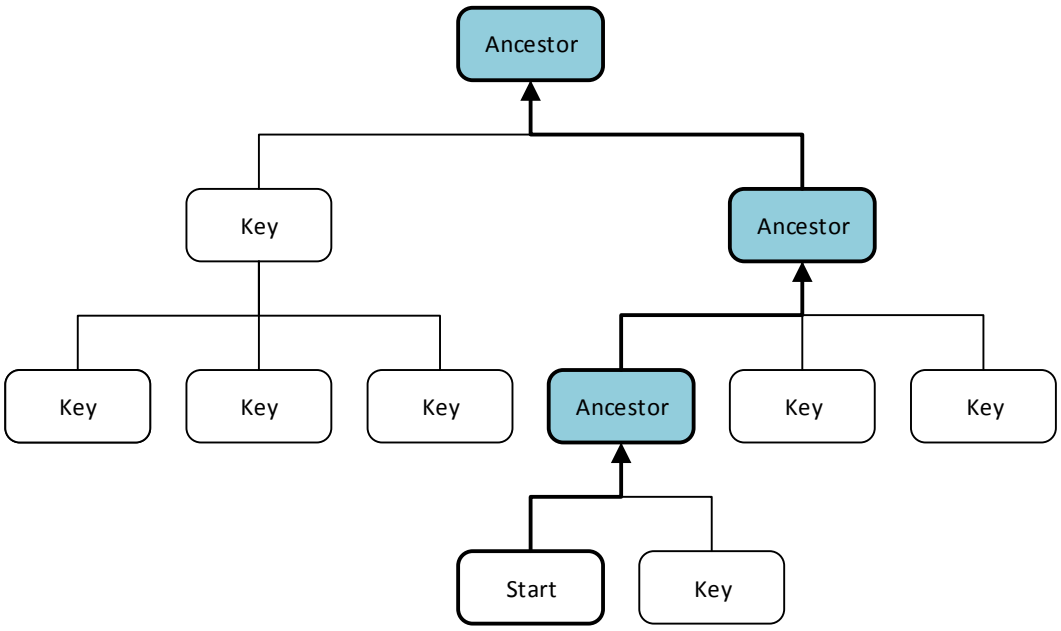
1.2

Return-type

Key

Description

This function returns all keys above the argument in their hierarchy (including the parents, the parent's parents and so on, up to the dimension root-key).



If the argument is NULL, the return value is also NULL.

Examples

```
ANCESTORS( Time:'01.07.2003' )
```

Returns Time:'Jul/2003' + Time:'2003' + Time:'Complete Period'

See also

PARENT, PEDIGREE

AND

Syntax

```
<and-expression> := 'AND(' value1: <Boolean> ',' value2:
<Boolean> ')'

<and-expression> := <Boolean> 'AND' <Boolean>
```

Since

1.0

Return-type

Boolean

Description

The logical operator AND returns one of the boolean constants TRUE or FALSE or NULL, depending on the arguments:

- If both arguments are TRUE, the result is also TRUE.
- If one of the arguments are FALSE, the result is also FALSE.
- If one of the arguments is NULL, the result is also NULL.

Instead of this function, you also can use the operator "AND".

Examples

```
AND( TRUE, TRUE )
```

= TRUE

```
TRUE and TRUE
```

= TRUE

```
AND( FALSE, TRUE )
```

= FALSE

```
AND( TRUE, FALSE )
```

= FALSE

```
AND( TRUE, NULL )
```

= NULL

```
AND( FALSE, NULL )
```

= NULL

```
AND( NULL, NULL )
```

= NULL

See also

NOT, OR

ATTRIBUTENAMES

Syntax

```
<attributes-expression> := 'ATTRIBUTES(' keys: <Key> ')'
```

Since

2.0

Return-type

String

Description

This function returns the names of all attributes of all keys passed as argument.

If the argument is NULL, the function returns NULL.

Examples

```
ATTRIBUTENAMES( Time:'01.07.2003' )
```

I.e. returns 'Weekday' | 'DayID' + ...

See also

ATTRIBUTENV, ATTRIBUTES, DIMENSIONATTRIBUTENAMES

ATTRIBUTENV

Syntax

```
<attributenv-expression> := 'ATTRIBUTENV(' keys: <Key> ')'
```

Since

2.0

Return-type

String

Description

This function returns a list containing all attribute-names and -values for the given keys. All values of a single attribute are returned as comma-separated strings. If the attribute has more than five values, the result is truncated.

If the argument is NULL, the function returns NULL.

Examples

```
ATTRIBUTENV( Product:Product1 )
```

I.e. returns 'ProductID' | 'P1' | 'Customer' | 'C1, C2, C3, C4, C5...' + ...

See also

ATTRIBUTENAMES, ATTRIBUTES, DIMENSIONATTRIBUTENAMES

ATTRIBUTES

Syntax

```
<attributes-expression> := 'ATTRIBUTES(' keys: <Key> ')'
```

Since

2.0

Return-type

Value

Description

This function returns all values of all attributes of the keys passed as argument.

If the argument is NULL, the function returns NULL.

Examples

```
ATTRIBUTES( Time:'01.07.2011' )
```

Returns Weekday:'Tue' + Week:'26/2011' + ...

See also

ATTRIBUTENAMES, ATTRIBUTENV, DIMENSIONATTRIBUTENAMES

AVG

Syntax

```
<avg-expression> := 'AVG(' values: <Number>, { ',' values:
<Number> } ')'
```

Since

1.0

Return-type

Double

Description

The function AVG returns the average value of all values of all arguments.

The function ignores NULL values - If you don't want to ignore NULL-values, you can use the ZERO function to interpret them as zero.

If the arguments contain no values or only NULL-values, the function returns an empty result.

Examples

```
AVG( 2, 4 )
```

= 3

```
AVG( 2, 4, NULL )
```

= 3

```
AVG( NULL )
```

= EMPTY

```
AVG( Amount( LEVEL( Time, 3 ) )
```

Average amount of all days (level 3 of time-dimension)

```
AVG( ZERO( Amount( LEVEL( Time, 3 ) ) )
```

Average amount of all days (level 3 of time-dimension), interpreting NULL as 0

```
AVG( NULL )
```

= NULL

See also

MAX, MIN, ZERO

AVGKEY

Syntax

```
<avgkey-expression> := 'AVGKEY(' keys: <Key> ',' expression:
<Value> ')'
```

Since

2.2.6

Return-type

Key

Description

The AVGKEY function evaluates the expression passed as the second argument for all keys of the first argument and returns the key with the result for being the nearest to the average value of all results.

If any of the arguments is NULLthe function returns NULL.

Examples

```
AVGKEY( PRODUCT, Amount() )
```

Returns the product with the closest amount to the average value over all products (for the current filter).

See also

MAXKEY, MINKEY

BEAUTIFY

Syntax

```
<beautify-expression> := 'BEAUTIFY(' text: <String> ')'
```

Since

2.0

Return-type

String

Description

The function BEAUTIFY changes the letters' case to make a string more readable. The first letter of each word will become uppercase. The rest of each word will be converted to lower case.

If you pass NULLto this function, it will return NULL, too.

Examples

```
BEAUTIFY( 'hello WORLD' )
```

= 'Hello World

```
BEAUTIFY( NULL )
```

= NULL

See also

LIMIT, TOLOWER, TOUPPER, XML2ASCII

BELONGSTO

Syntax

```
<belongsto-expression> := 'BELONGSTO(' keys: <Key> ')'
```

Since

2.2

Return-type

Boolean

Description

This function returns TRUE, if the currently filtered key (or at least one if more than one are in the filter) of the dimension defined by the arguments is equal, child of or parent of the key passed as the argument. If multiple keys are passed as argument, the function returns TRUE if at least one key belongs to the current filter.

If the argument is NULL, the function returns FALSE.

This function is mainly used in matching expression, for cubes in instance.

Examples

```
BELONGSTO( Time:'01/2005' )
```

Returns true for Time:'01.01.2005', Time:'2005' etc.

See also

HASKEYS

CATCH

Syntax

```
<catch-expression> := 'CATCH(' expression: <Any>, alternative
result: <Any> ')'
```

Since

2.5

Return-type

The common super-type of both arguments

Description

The CATCH function catches and suppresses any error thrown while the evaluation of the first arguments and returns the result of the second argument instead. If no error occurs, the result of the first argument is returned.

The return type of this function is the common super type of both arguments. For example, if the first argument has the type "Double" and the second has "String", the return type would be "Value".

Examples

```
CATCH( Amount(), 'Not available' )
```

Evaluates "Amount()" and returns (if no error was thrown) the result of this expression. If any error occurs (i.e.. if the fact Amount is not available offline and no database is connected), the string 'Not available' is returned.

See also

ERROR

CASE

Syntax

```
<case-expression> := 'CASE(' <Boolean> ',' <Any> { ','
<Boolean> ',' < Any> } ')'
```

Since

2.2.2

Return-type

The return-type is the super-type of all values.

Description

The CASE-function allows the conditional evaluation of one or more arguments. The CASE-function is very similar to the IIF function but accepts more than one condition.

The system begins with the first boolean condition. If the expression returns TRUE, the following argument is evaluated and returned as the result. If it returns FALSE, then the next condition is evaluated and so on.

If any of the evaluated conditions returns NULL, the function returns NULL. If no condition returns TRUE the function also returns NULL.

Because there is no default return-value for this CASE function, you can add TRUE as the last condition, followed by the default value. This condition will always be returned if no other condition matches.

Examples

```
CASE(
  HASLEVEL( Time, 1 ), 'L1',
  HASLEVEL( Time, 2 ), 'L2',
  TRUE, 'Other'
)
```

Returns 'L1' for years, 'L2' for months and 'Other' for any other level of the current time-selection.

```
CASE(
  HASLEVEL( Time, 1 ), 'L1',
  NULL, 'L2'
)
```

Returns 'L1' for years and NULL for any other level of the time selection.

See also

IIF, SWITCH

CEIL

Syntax

```
<ceil-expression> := 'CEIL(' value: <Number> ')'
```

Since

2.0

Return-type

Integer

Description

Returns the smallest (closest to negative infinity) integer that is not less than the argument. Special cases:

- If the argument value is of the type integer, then the result is the same as the argument.
- If the argument is NULL, then the result is also NULL.

Examples

```
CEIL( 1.5 )
```

= 2

```
CEIL( -1.5 )
```

= -1

```
CEIL( 5 )
```

= 5

```
CEIL( NULL )
```

= NULL

See also

FLOOR

CHILDREN

Syntax

```
<children-expression> := 'CHILDREN(' keys: <Key> ')'
```

Since

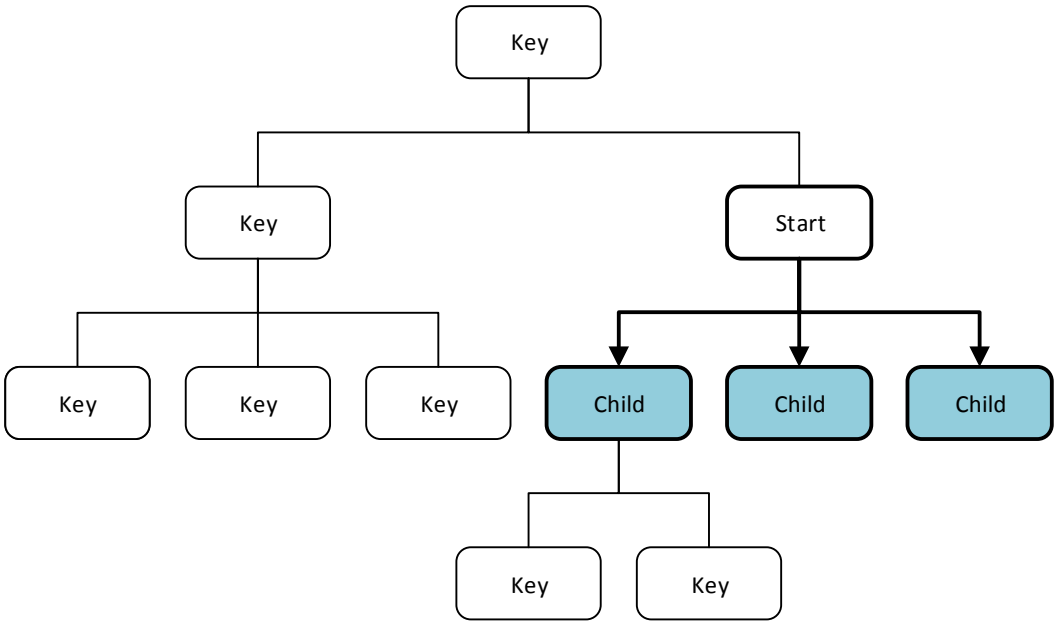
1.0

Return-type

Key

Description

The function CHILDREN returns all children of the keys passed as argument. Children of a key are all keys located directly under a key inside a hierarchy.



If the argument is NULL, this function returns NULL.

Examples

```
CHILDREN( Time:'2011' )
```

I.e. returns Time:'Jan/2011' + Time:'Feb/2011' + ... + Time:'Dec/2011'

```
CHILDREN( NULL )
```

= NULL

See also

ANCESTORS, FAMILY, LEAFS, NEIGHBOURS, PARENT, PEDIGREE

CLUSTER

Syntax

```
<cluster-expression> := 'CLUSTER(' keys: <Key> ')'
```

Since

2.1.2

Return-type

Key

Description

The function CLUSTER replaces keys in the argument with their parents if all siblings of the key also appear in the argument. All children of the parent are removed from the result then.

The result of the CLUSTER-function is clustered again unless it is not cluster-able any more.

This function is useful to reduce the number of loaded keys and facts from a database or store, for instance, if you want to aggregate a fact for a specified range of time.

If the argument is NULL, the function returns NULL.

Examples

```
CLUSTER( Time:'Q1/2010' | Time:'Q2/2010' | Time:'Q3/2010' |
Time:'Q4/2010' | Time:'Q1/2011' ) )
```

= Time:'2010' + Time:'Q1/2011'

```
CLUSTER( NULL )
```

= NULL

```
SUM( Amount( CLUSTER( Time ) ) )
```

Efficient aggregation of the fact "Amount" for a time-span

See also

RANGE

COALESCE

Syntax

```
<coalesce-expression> := 'COALESCE( values: <Any> { ',' values: <Any> } )'
```

Since

2.6.0

Return-type

The common super-type of all arguments

Description

This function returns the first value from all arguments which is not NULL. If all arguments are NULL, the function will return NULL.

Examples

```
COALESCE( NULL, NULL, 1, 5 )
```

= 1

```
COALESCE( Person.privateAddress, Person.businessAddress )
```

Returns the private address, or the business address if the private does not exist.

See also

EXISTS, ZERO

COLSPAN

Syntax

```
<colspan-expression> := 'COLSPAN(' y: <Integer> ')'
```

Since

2.2.2

Return-type

Integer

Description

This function returns the number of columns spanned by the header in the same column with the given Y position (passed as argument). I.e. COLSPAN(0) returns the span of the topmost header.

For instance, you can use this function to create subtotals for tables with grouped headers.

Examples

```
SUM( TONUMBER( MATRIX( X() - 1, Y(), X() - COLSPAN(0), Y() ) ) )
```

Returns a subtotal for all columns spanned by the same, grouping header in the X axis.

See also

ROWSPAN

CONCAT

Syntax

```
<concat-expression> := 'CONCAT(' text: <String> [ ',' delimiter: <String> ] ')'
```

Since

2.0

Return-type

String

Description

The function CONCAT concatenates all strings from the first argument to one string, using the delimiter defined with the optional second argument. If no delimiter is defined, the standard-delimiter "," is used.

If the first argument is NULL, this function returns NULL.

Examples

```
CONCAT( LEVEL( Customer, 1 ).Name )
```

I.e. returns 'Customer1,Customer2,Customer3'

```
CONCAT( LEVEL( Customer, 1 ).Name, ';' )
```

I.e. returns 'Customer1;Customer2;Customer3'

See also

ADD, SPLIT

CONTAINS

Syntax

```
<contains-expression> := 'CONTAINS(' values: <Any> ',' search-value: <Any> ')'
```

Since

1.2

Return-type

Boolean

Description

The function CONTAINS returns TRUE, if all values of the second argument are contained in the first argument, otherwise it returns FALSE. NULL is treated like any other element.

Examples

```
CONTAINS( FAMILY( Time:'Complete Period' ), LEVEL( Time, 1 ) )
```

= TRUE

```
CONTAINS( FAMILY( Time:'Complete Period' ), 'Hello World' )
```

= FALSE

```
CONTAINS( FAMILY( Time:'Complete Period' ), NULL )
```

= NULL

```
CONTAINS( NULL, LEVEL( Time, 1 ) )
```

= FALSE

See also

IN

CONTAINSTEXT

Syntax

```
<containstext-expression> := 'CONTAINSTEXT(' text: <String>
',' search-text: <String> ')'
```

Since

2.2

Return-type

Boolean

Description

Returns TRUE, if the text passed as first argument contains the text defined by the second argument. If any of the arguments is NULL, the function returns NULL. The text-comparison is case-sensitive.

Examples

```
CONTAINSTEXT( 'Hello world', 'Hello' )
```

= TRUE

```
CONTAINSTEXT( 'Hello world', 'Foo' )
```

= FALSE

```
CONTAINSTEXT( 'Hello world', 'World' )
```

= FALSE

```
CONTAINSTEXT( 'Hello world', NULL )
```

= NULL

```
CONTAINSTEXT( NULL, 'Hello' )
```

= NULL

See also

TEXTPOSITION

COUNT

Syntax

```
<count-expression> := 'COUNT(' <Any> ')'
```

Since

1.2

Return-type

Integer

Description

This function returns the size of the argument. If the argument is NULL, this function returns NULL.

Examples

```
COUNT( LEVEL( Fact, 0 ) )
```

= 1 (the root-key)

```
COUNT( NULL )
```

= NULL

See also

EXISTS

COUNTRY

Syntax

```
<country-expression> := 'COUNTRY()'
```

Since

2.2.0

Return-type

String

Description

This function returns the country-code of the current session locale. The session locale is usually defined and transferred by the Browser.

Examples

```
COUNTRY()
```

= 'DE'

See also

LANGUAGE, LOCALE

CUBE

Syntax

```
<cube-expression> := 'CUBE(' [ filter-modifyer: <Key> { ',' filter-modifyer: <Key> } ] ')'

<cube-expression> := '[' [ <Key> { ',' <Key> } ] ']'

<cube-expression> := <fact-name> '(' [ <Key> { ',' <Key> } ] ')'
```

Since

1.0

Return-type

The type of the returned fact (Double, Integer, Boolean or Date)

Description

With the CUBE-function, you can read data out of one or more cubes. Without any argument, this function will read and return the current selection from the cubes. The current selection is defined by the headers, query- and block-filters, selectors, etc.

You could, as an example, define a query with an x-header containing the key Fact:Amount and a y-header containing the key Product:Product1. You also could add a selector with years as option. If the user selects Time:2003, the current selection for the cells is Fact:Amount, Product:Product1, Time:2003. If you use the CUBE-function inside the cells, this value will be read out of the cube.

If you want to read values, which differ from the current selection, you can pass key-expressions as arguments to the CUBE-function. The results of this key-expression are to the selection before reading values. If you used the expression “CUBE(NEXT(Time))” in the above sample, the same value (for Fact:Amount and Product:Product1) would be accessed, but for the next year. Alternatively, you could use “CUBE(Fact:Price)” to access another fact (the price in this case).

Note that the CUBE-function can return more than one value. This happens if the current selection or the arguments contain multiple keys for a least one dimension. If you used “CUBE(CHILDREN(Time))” in the above sample, it would return 12 values, one per month.

Mathematically spoken, the number of the returned values is (N1) * (N2) * ... * (Nd), with Nx being the number of selected keys for dimension x and d being the number of possible dimensions.

Return type

The return-type of this function depends on the selected fact. If exactly one fact is selected, its type will be the return-type. If more than one fact is selected, the most common super type of their return-types will be returned (Any in the worst case).

Short forms

There are two short-forms of this function: Brackets and auto-generated fact-functions. The bracket-form is very close to the original-form - instead of using the function-name CUBE you can enclose all arguments in brackets []. The function ”CUBE(NEXT(Time))” for example, is equal to “[NEXT(Time)]”.

The auto-generated fact-functions can be used whenever you exactly know which fact you want to read (which you normally know). In this case, you can use the function which is automatically created and named like your fact. For instance. if you define a fact named "Amount" there will be a function called "Amount". In this case, you can access the fact with Amount() instead of “[Fact:Amount]”. You also can pass key-expressions to the generated fact-function. A valid example would be “Amount(NEXT(Time))”.

Note that only well-named fact will result in an auto-generated function. If your fact contains special chars of white spaces, the system won't be able to create a function for this.

Examples

```
CUBE()
```

Returns all values for the current filter

```
[]
```

Same as above

```
CUBE( NEXT( Time ) )
```

Returns the selected values, but for the next time-element

```
[ NEXT( Time ) ]
```

Same as above

```
[ NEXT( Time ), Fact:Amount ]
```

Returns the value of "Amount" for the next time-element

```
CUBE( NEXT( Time ), Fact:Amount )
```

Same as above

```
Amount( NEXT( Time ) )
```

Same as above

See also

LOOKUP

aa**DATEADD**

Syntax

```
<dateadd-expression> := 'DATEADD(' date: <Date> ',' amount:
<Integer> [ ',' type: <String> ] ')'
```

Since

2.5

Return-type

Date

Description

The DATEADD function adds years, days, hours, minutes, seconds or milliseconds to the date passed as first argument. The number of units is defined by the second argument.

The last and optional argument defines, what you want to add to the date. You can pass a string 'year', 'day', 'hour', 'minute', 'second' or 'millisecond' here. The default value for the last argument is 'day'.

There is no function to sub units from a date. Instead, you can pass a negative value as second argument.

If any of the arguments is NULL the function returns NULL.

Examples

```
DATEADD( Product.releaseDate, 5 )
```

Adds 5 days to the release date of the product.

```
DATEADD( Product.releaseDate, -1, 'year' )
```

Subtracts 1 year from the release date of the product.

See also

DATEDIFF

DATEDIFF

Syntax

```
<datediff-expression> := 'DATEDIFF(' date1: <Date> ',' date2:
<Date> ')'
```

Since

2.5

Return-type

Integer

Description

The DATEDIFF function returns the difference between both arguments in milliseconds.

If any of the arguments is NULL, the function returns also NULL.

Examples

```
DATEDIFF( Project.End, Project.Start ) / ( 24 * 60 * 60 * 1000
)
```

Calculates a project duration and converts it in days.

See also

DATEADD

DEBUG

Syntax

```
<debug-expression> := 'DEBUG(' expression: <Any> [ ',' debug-
text: <String> ] ')'
```

Since

2.2.1

Return-type

The type of the first argument

Description

This function is useful to debug calculations in formulas or queries. It returns the value of the first argument but also prints the values (together with the text defined by the optional second argument) to the debug log.

If no debug text is given, the default text “DEBUG:” is used.

Examples

```
DEBUG( PARENT( Time ), 'PARENT OF TIME:' )
```

Returns PARENT(Time) and prints the result to the debug log, for instance:

```
"PARENT OF TIME: 2006"
```

See also

FILTER, RETURNTYPE, TYPE

DEFAULTTEXT

Syntax

```
<defaulttext-expression> := 'DEFAULTTEXT(' keys: <Any> ')'
```

Since

2.2.1

Return-type

String

Description

This function returns the default display-text of the argument. This only affect keys. All other values will be simply converted to string.

If the key dimension has a default-text attribute, then the value of this attribute is returned. If the dimension has not default text-attribute or the key does not contain any value for this attribute, the ID of the key is returned.

The DEFAULTTEXT function exactly returns the same text pivot-tables display for keys in headers by default.

If the argument is NULL, the function also returns NULL.

Examples

```
DEFAULTTEXT( Product:1234 )
```

i.e. = 'Computers'

```
DEFAULTTEXT( NULL )
```

= NULL

DEVIATION

Syntax

```
<deviation-expression> := 'DEVIATION(' value1: <Number> ',' value2: <Number> ')'
```

Since

2.2

Return-type

Number

Description

It calculates the deviation between the first and the second argument in percent (where 0 means 0%, 1 means 100%, -1 means -100%):

- If any of the arguments is NULL, the function returns 0.
- If both values are equal, the function returns 0.

- If the first argument is 0 and the second is positive, the function returns 1.
- If the first argument is 0 and the second is negative, the function returns -1.
- Otherwise the function returns (v2 - v1) / v1 where v1 is the value defined by the first argument and v2 the value defined by the second.

Examples

```
DEVIATION( 10, 20 )
```

= 1

```
DEVIATION( 20, 10 )
```

= -0.5

```
DEVIATION( 0, 20 )
```

= 1

```
DEVIATION( 0, -20 )
```

= -1

```
DEVIATION( 20, 20 )
```

= 0

```
DEVIATION( 0, 0 )
```

= 0

```
DEVIATION( NULL, 20 )
```

= NULL

```
DEVIATION( 10, NULL )
```

= NULL

See also

DIV, PERCENTILE

DEPTH

Syntax

```
<depth-expression> := 'DEPTH(' dimension: <Key> ')'
```

Since

2.1

Return-type

Integer

Description

This function returns the depth (the number of levels) of the dimension specified by the argument. If the argument is NULL, the function returns NULL.

Examples

```
DEPTH( Time )
```

= 4

```
DEPTH( NULL )
```

= NULL

See also

LEVELNAMES

DIMENSIONATTRIBUTENAMES

Syntax

```
<dimensionattributenames-expression> :=
'DIMENSIONATTRIBUTENAMES(' dimension: <Key> ')'
```

Since

2.1

Return-type

String

Description

This function returns the names of all attributes of all keys of the argument. If the argument is NULL, the function returns NULL.

Examples

```
DIMENSIONATTRIBUTENAMES( Product )
```

I.e. returns 'ProductID' + 'Customer' + 'Text' + ...

See also

ATTRIBUTENAMES, ATTRIBUTENV, ATTRIBUTES

DIMENSIONCAPTION

Syntax

```
<dimensioncaption-expression> := 'DIMENSIONCAPTION('
dimension: <Key> ')'
```

Since

3.1

Return-type

String

Examples

```
DIMENSIONCAPTION(Product)
```

I.e. returns “Company Products” if the caption was “Company Product”.

See also

DIMENSIONNAME

DIMENSIONNAME

Syntax

```
<dimensionname-expression> := 'DIMENSIONNAME(' keys: <Key> ')'
```

Since

2.0

Return-type

String

Description

Returns the names of the dimension(s) of the keys passed as argument. If the argument contains keys from different dimension, the function returns all dimension-names. If the argument is NULL, it returns NULL.

Note that this function does not execute the argument, it only analyses its return type and converts it into dimension names.

Examples

```
DIMENSIONNAME( Product )
```

= 'Product'

```
DIMENSIONNAME( Product:Product1 )
```

= 'Product'

```
DIMENSIONNAME( FIRST( Product::1 | Time::1 ) )
```

= 'Product' | 'Time'

See also

DIMENSIONCAPTION, DIMENSIONNAMES, LEVELNAMES

DIMENSIONNAMES

Syntax

```
<dimensionnames-expression> := 'DIMENSIONNAMES()'
```

Since

2.0

Return-type

String

Description

Returns the names of all dimensions of the current model.

Examples

```
DIMENSIONNAMES()
```

I.e. returns 'Fact' + 'Product' + 'Time' + ...

See also

DIMENSIONCAPTION, DIMENSIONNAME, LEVELNAMES

DISTINCT

Syntax

```
<distinct-expression> := 'DISTINCT(' <Any> ')'
```

Since

2.0

Return-type

Equal to the argument type.

Description

The DISTINCT-function eliminates double values from argument. Each value which occurs more than once in the argument will appear only once in the result. This is also valid for NULL.

Examples

```
DISTINCT( Product:Product1 )
```

= Product:Product1

```
DISTINCT( Product:Product1 | Product:Product2 |
Product:Product1 )
```

= Product:Product1 + Product:Product2

DIV

Syntax

```
<div-expression> := 'DIV(' value: <Number> ',' divisor:
<Number> ')'

<div-expression> := <Number> '/' <Number>
```

Since

1.0

Return-type

Double

Description

The function DIV divides every value from the first argument by the value of the second argument and returns a number of doubles. The argument can only have a single value.

If one of both values is NULL, the result is NULL. If the divisor is zero, the function returns the string "ERR".

Instead of this function, you also can use the operator "/".

Examples

```
DIV(10, 2)
```

= 5

```
10 / 2
```

= 5

```
DIV(10, NULL)
```

= NULL

```
DIV( NULL, 5 )
```

= NULL

```
DIV( 10, 0 )
```

= 'ERR'

```
DIV( JOIN( 10, 5 ), 2 )
```

= 5 | 2.5

See also

MOD, MUL

DRILLKEY

Syntax

```
<drillkey-expression> := 'DRILLKEY()'
```

Since

2.2.1

Return-type

Key

Description

DRILLKEY returns the current drill-key of a header. This is the iteration-key of the header generated by a drill-down.

This function is only valid in pivot-tables.

Examples

```
DRILLKEY()
```

In a normal drill-down using the children hierarchy, for instance, the time-dimension, the drill-key can i.e. be. 'Time:2006' for the header containing 'Time:Jan/2006'.

See also

DRILLLEVEL, ITERATIONKEY

DRILLLEVEL

Syntax

```
<drilllevel-expression> := 'DRILLLEVEL()'
```

Since

2.1

Return-type

Integer

Description

Returns the current drill-level of a header. The level for a not-drilled header is 0, after a single drill-down 1 and so on. Headers without drill-down -feature always return 0.

This function is only valid in pivot-tables.

Examples

```
DRILLLEVEL()
```

Returns 0 if no drill-down happened or 1 if the user drilled once

See also

DRILLKEY, ITERATIONKEY

DURATIONTOSTRING

Syntax

```
<durationtostring-expression> := 'DURATIONTOSTRING(
milliseconds: <Integer> )'
```

Since

2.6.1

Return-type

String

Description

This function formats the passed duration (in milliseconds) into a string. The standard format for durations is "days-HH:MM:SS". If the argument is NULL, the function returns NULL.

Examples

```
DURATIONTOSTRING( 90000 )
```

= "00:01:30"

See also

TOSTRING

ELEMENT_AT

Syntax

```
<elementat-expression> := 'ELEMENT_AT(' values: <Any> ','
position: <Integer> [ ',' count: < Integer> ] ')'
```

Since

2.2.2

Return-type

The type of the first argument

Description

This function returns an element from the first argument for the given position. The position is defined by the second argument whereby 0 is the position of the first value.

If more than one element should be returned, you can define the number of elements with the optional third argument.

If any of the arguments is NULL, then the function also returns NULL.

Examples

```
ELEMENT_AT( 'A' | 'B' | 'C' , 0 )
```

= 'A'

```
ELEMENT_AT( 'A' | 'B' | 'C', 2 )
```

= 'C'

```
ELEMENT_AT( 'A' | 'B' | 'C', 1, 2 )
```

= 'B' | 'C'

```
ELEMENT_AT( 'A' | 'B' | 'C', NULL )
```

= NULL

See also

CONTAINS, IN

EMPTY

Syntax

```
<empty-expression> := 'EMPTY()'
```

Since

2.1

Return-type

All

Description

This function returns an empty list. Because the return-type of this function is "All" you can use it as argument for any other function.

Examples

```
EMPTY()
```

Returns an empty list.

See also

COUNT, EMPTYASNULL, ISEMPTY

EMPTYASNULL

Syntax

```
<emptyasnull-expression> := 'EMPTYASNULL( value: <Any> )'
```

Since

2.6.1

Return-type

The type of the argument

Description

If the argument is an empty list, this function returns NULL. Otherwise the function returns the unmodified argument.

Examples

```
EMPTYASNULL( 10 | 20 )
```

= 10 | 20

```
EMPTYASNULL( EMPTY() )
```

= NULL

See also

EMPTY, ISEMPTY

ENDSWITH

Syntax

```
<endswith-expression> := 'ENDSWITH(' text: <String>, search-
text: <String> ')'
```

Since

2.1

Return-type

Boolean

Description

This function tests if the first argument ends with the string passed as second argument. If any of the arguments is NULL, the function returns NULL.

Examples

```
ENDSWITH( 'Hello World', 'World' )
```

= TRUE

```
ENDSWITH( 'Hello World', 'Hello' )
```

= FALSE

```
ENDSWITH( 'Hello World', 'NULL' )
```

= NULL

See also

CONTAINSTEXT, STARTSWITH

EQUAL

Syntax

```
<equal-expression> := 'EQUAL(' values: <Any> ',' values: <Any>
')'

<equal-expression> := <Any> '=' <Any>
```

Since

1.0

Return-type

Boolean

Description

This function tests both arguments for equality. They are equal if:

- Both arguments have the same size AND
- each element of argument one is contained in argument two at the same position.

Instead of this function, you also can use the operator "=".

Examples

```
EQUAL( 10, 10 )
```

= TRUE

```
EQUAL( 10, 20 )
```

= FALSE

```
EQUAL( 10 | 10, 10 )
```

= FALSE

```
EQUAL( 10 | 20, 10 | 20 )
```

= TRUE

```
EQUAL( 10, 'Hello World' )
```

= FALSE

```
EQUAL( 10, NULL )
```

= NULL

```
EQUAL( NULL, NULL )
```

= NULL

See also

GREATER, GREATER_OR_EQUAL, LESS, LESS_OR_EQUAL, LIKE, UNEQUAL

ERROR

Syntax

```
<error-expression> := 'ERROR( message: <String > )'
```

Since

2.2.0

Return-type

All

Description

This function stops the current query-execution and raises an error with the error message passed as the argument. The error will be visible for the user.

If the argument is NULL, no error will be raised.

Examples

```
IIF( LEVELOF( Time ) = 0, ERROR( 'Invalid time-selection' ),
Time
```

See also

CATCH

EVAL

Syntax

```
<eval-expression> := 'EVAL(' expression: <String> ')'
```

Since

2.0

Return-type

Any

Description

The function EVAL allows to parse and execute expressions at runtime. The argument must be a string defining a valid expression. If the argument is NULL, this function returns NULL.

Because the system cannot know the return-type of the function at the compile time, its return-type is 'Any'. Usually, you must cast the result to the desired type to use the result of the function. Use one of the functions TODATE, TOINTEGER, TOKEY or TOSTRING for the cast.

Examples

```
TOINTEGER( EVAL( '10 * 2' ) )
```

= 5

```
EVAL( $param )
```

Parsing and evaluation of the parameter named “param”.

```
EVAL( 'FIND( $dimension, $id )' )
```

Find keys in the dimension defined by parameter “$dimension”.

```
EVAL( NULL )
```

= NULL

EXISTS

Syntax

```
<exists-expression> := 'EXISTS(' values: <Any> ')'
```

Since

1.1

Return-type

Boolean

Description

This function tests if the argument contains at least one other value than NULL.

Examples

```
EXISTS( NEXT( Time ) )
```

Returns TRUE, if "Time" not currently selects the last element.

```
EXISTS( 1 )
```

= TRUE

```
EXISTS( NULL )
```

= FALSE

See also

COUNT

EXP

Syntax

```
<exp-expression> := 'EXP(' value: <Number> ')'
```

Since

2.1

Return-type

Number

Description

This function returns Euler's number e raised to the power defined by the argument. If the argument is NULL, the function returns NULL.

Examples

```
EXP( 10 )
```

= e^10

See also

LOG, POW

FACTROOT

Syntax

```
<factroot-expression> := 'FACTROOT()'
```

Since

1.2

Return-type

Key

Description

The function "FACTROOT" returns the root-key of the fact-dimension.

Examples

```
FACTROOT()
```

E.g. returns Fact:'All Facts'

See also

NONFACTROOTS, UNIT

FAMILY

Syntax

```
<family-expression> := 'FAMILY(' keys: <Key> ')'
```

Since

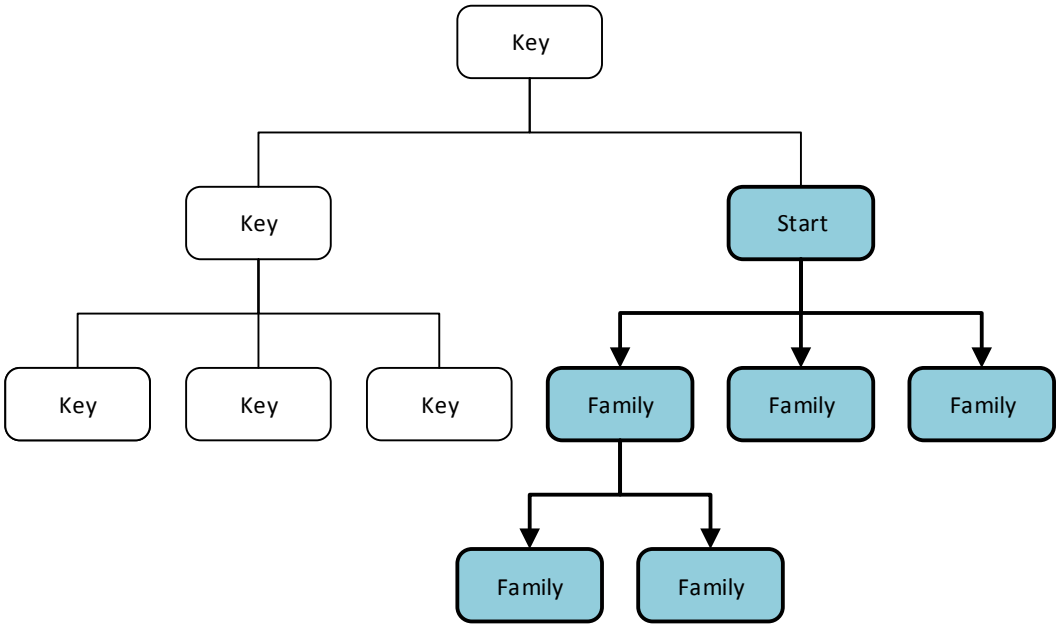
1.2

Return-type

Key

Description

The FAMILY function returns the family for each key of the argument and returns it. The members of a key-family are the key itself, its children, the children of the children and so on - down to the leafs.



If the argument is NULL, the functions returns NULL.

Examples

```
FAMILY( Time:'2011' )
```

I.e. returns Time:'Jan/2011' + Time:'01.01.2011' + ... + Time:'31.12.2011'

```
FAMILY( NULL )
```

= NULL

See also

ALL, ANCESTORS, CHILDREN, LEAFS, NEIGHBOURS, PARENT, PADIGREE

FILTER

Syntax

```
<filter-expression> := 'FILTER(' [ show-roots: <Boolean> ] ')'
```

Since

2.0

3.0 (argument “show-roots” added)

Return-type

String

Description

The function FILTER returns the current filter as string. The current filter consists of all dimension-selections, i.e. in a cell.

The optional boolean argument determines if selected root keys are included in the result string. By default, all roots are included.

This is a debug-function, which you can use to test and display the current selection if needed.

Examples

```
FILTER()
```

I.e. returns the string '[Time=2003;Product=A,B;Customer=All customers]'

```
FILTER( FALSE )
```

I.e. returns the string '[Time=2003;Product=A,B]'

See also

DEBUG, FILTERKEYS

FILTERKEYS

Syntax

```
<filterkeys-expression> := 'FILTERKEYS()'
```

Since

2.2.6

Return-type

Key

Description

The function FILTERKEYS returns all keys of the current filter as keys. The current filter consists of all dimension-selections, i.e. in a cell.

Examples

```
FILTERKEYS()
```

E.g. returns the keys Time:2003 | Product:A | Product:B | Customer:All

See also

FILTER

FIND

Syntax

```
<find-expression> := 'FIND( dimension: <Key> ',' pattern:
<String> [ ',' use-wildcards: <Boolean> [ ',' search-
attribute: < String> ] ] ')'
```

Since

2.0

2.5 (attribute name added)

Return-type

Key

Description

The function FIND searches for keys of the dimension defined by the first argument using the pattern defined by argument two.

The third and optional argument determines if you want to use wild-cards when searching for keys. When enabling wild-cards, you can use the characters '*' and '?' to find more than one key. The default value for this argument is "false".

The last argument defines the name of the attribute you want to search. If the value is NULL, the function will return all keys which IDs match the pattern. If you define a valid attribute name, the function will return all keys which have values for this attribute, which match the pattern. By default the FIND function searches for IDs.

If one of the first three arguments is NULL, the FIND-function returns NULL.

Examples

```
FIND( Time, 'Jan/2011' )
```

E.g. returns Time:'Jan/2011'

```
FIND( Time, 'J*/2003', true )
```

E.g. returns Time:'Jan/2011', Time:'Jun/2011', Time:'Jul/2011'

```
FIND( Time, NULL )
```

= NULL

```
FIND( NULL, 'J*/2011', true )
```

= NULL

```
FIND( Time, '55', false, 'Week' )
```

I.e. returns all Time keys which have a value for the "Week" attribute equals to "55".

See also

MATCH, NOW

FIRST

Syntax

```
<first-expression> := 'FIRST(' values: <Any> [ ',' size:
<Integer> ] ')'
```

Since

1.2

Return-type

The type of the first argument.

Description

This function returns the first N values of the first argument. If you define no size N (with the second argument), the function only returns the first element of argument.

If any of the arguments is NULL, the function returns NULL.

Examples

```
FIRST( LEVEL( Time, 2 ) )
```

E.g. returns Time:'Jan/2003'

```
FIRST( LEVEL( Time, 2 ), 3 )
```

E.g. returns Time:'Jan/2003' | Time:'Feb/2003' | Time:'Mrz/2003'

```
FIRST( NULL )
```

= NULL

See also

LAST, NEXT, PREV

FKEY

Syntax

```
<fkey-expression> := 'FKEY()'
```

Since

2.5.0

Return-type

Key

Description

The FKEY function returns the last key added to the filter. In headers, this is the current key of the header iteration. In FOREACH, MATCH, SORT or similar expressions, this is the iterated key.

Examples

```
FOREACH( PRODUCT | MANUFACTURER, {FKEY()} )
```

See also

FOREACH, IVALUE, MATCH

FLOOR

Syntax

```
<floor-expression> := 'FLOOR(' value: <Number> ')'
```

Since

2.0

Return-type

Integer

Description

This function returns the largest (closest to positive infinity) integer value that is not greater than the argument. If the argument is NULL, it returns NULL.

Examples

```
FLOOR( 1.5 )
```

= 1

```
FLOOR( -1.5 )
```

= -2

```
FLOOR( 5 )
```

= 5

```
FLOOR( NULL )
```

= NULL

See also

CEIL, ROUND

FOREACH

Syntax

```
<foreach-expression> := 'FOREACH(' values: <Any> ',' 
expression: <Any> ')'
```

Since

1.2, changed in 2.5.0

Return-type

The return-type of the second argument.

Description

The FOREACH function evaluates the second argument for each value of the first argument and returns all results in a joined list. If one of the arguments is NULL, this function returns NULL.

At the beginning, the first argument is evaluated - which results in a list of keys or values.

If the first argument returns keys, they are added to the current filter, and the second argument is evaluated with this filter.

If the first argument does not return keys, the function will be executed with the original filter, and you can use the IVALUE function to access the iterated value.

Examples

```
FOREACH( LEVEL( Article, 2 ), Article.Color )
```

Returns all colors of all articles at level 2

See also

FKEY, IVALUE, MATCH

FORECAST

Syntax

```
<forecast-expression> := 'FORECAST('
  expression: <Number>
  ',' input-key:s <Key>
  ',' output-key: <Key>
  [ ',' model: <Integer>
  [ ',' periodicity: <Integer> ]
  [ ',' iterations: <Integer> ] ] ] ')'
```

Since

2.2

Return-type

Double

Description

The FORECAST function performs a forecast for the function defined with argument 1 for a single dimension-element (defined by argument 3) based on the results for the keys defined by the argument 2.

This function offers a number of different forecast-models, from which you can choose one with the 4th argument. This argument expects an integer-number representing one of the following forecast-models:

- **0: Best forecast (default):** This option chooses automatically one the following models.

- **1: Simple exponential forecast:** A simple exponential smoothing forecast model is a very popular model used to produce a smoothed Time Series. Whereas in simple Moving Average models the past observations are weighted equally, Exponential Smoothing assigns exponentially decreasing weights as the observations get older.

- **2: Double exponential forecast:** Double exponential smoothing - also known as Holt exponential smoothing - is a refinement of the popular simple exponential smoothing model but adds another component which takes into account any trend in the data. Simple exponential smoothing models work best with data where there are no trend or seasonality components to the data. When the data exhibits either an increasing or decreasing trend over time, simple exponential smoothing forecasts tend to lag behind observations. Double exponential smoothing is designed to address this type of data series by taking into account any trend in the data.

Note that double exponential smoothing still does not address seasonality. For better exponentially smoothed forecasts using data where there is expected or known to be seasonal variation in the data, use triple exponential smoothing.

- **3: Triple exponential forecast:** Triple exponential smoothing - also known as the Winters method - is a refinement of the popular double

exponential smoothing model but adds another component which takes into account any seasonality - or periodicity - in the data.

• **4: Regression forecast:** Implements a single variable linear regression model. The coefficients of the regression - the intercept and the slope - as well as the accuracy indicators are determined from the data set passed to initialize.

A single variable linear regression model essentially attempts to put a straight line through the data points. For the more mathematically inclined, this line is defined by its gradient or slope, and the point at which it intercepts the x-axis (i.e. where the independent variable has, perhaps only theoretically, a value of zero).

• **5: Polynomial forecast:** Implements a single variable polynomial regression mode. The coefficients of the regression as well as the accuracy indicators are determined from the data set passed to initialize.

A single variable polynomial regression model essentially attempts to put a polynomial line - a curve if you prefer - through the data points.

• **6: Multiple linear forecast:** Implements a multiple variable linear regression model. The coefficients of the regression, as well as the accuracy indicators are determined from the data set passed to initialize.

A multiple variable linear regression model essentially attempts to put a hyperplane through the data points.

• **7: Moving average forecast:** A moving average forecast model is based on an artificially constructed time series in which the value for a given time period is replaced by the mean of that value and the values for some number of preceding and succeeding time periods. As you may have guessed from the description, this model is best suited to time-series data; i.e. data that changes over time. For example, many charts of individual stocks on the stock market show 20, 50, 100 or 200 day moving averages as a way to show trends.

Since the forecast value for any given period is an average of the previous periods, then the forecast will always appear to "lag" behind either increases or decreases in the observed (dependent) values. For example, if a data series has a noticeable upward trend then a moving average forecast will generally provide an underestimate of the values of the dependent variable.

The moving average method has an advantage over other forecasting models in that it does smooth out peaks and troughs (or valleys) in a set of observations. However, it also has several disadvantages. In particular this model does not produce an actual equation.

Therefore, it is not all that useful as a medium-long range forecasting tool. It can only reliably be used to forecast one or two periods into the future.

The moving average model is a special case of the more general weighted moving average. In the simple moving average, all weights are equal.

• **8: Weighted moving average forecast:** A weighted moving average forecast model is based on an artificially constructed time series in which the value for a given time period is replaced by the weighted mean of that value and the values for some number of preceding time periods. As you may have guessed from the description, this model is best suited to time-series data; i.e. data that changes over time.

Since the forecast value for any given period is a weighted average of the previous periods, then the forecast will always appear to "lag" behind either increases or decreases in the observed (dependent) values. For example, if a data series has a noticeable upward trend then a weighted moving average forecast will generally provide an underestimate of the values of the dependent variable.

The weighted moving average model, like the moving average model, has an advantage over other forecasting models in that it does smooth out peaks and troughs (or valleys) in a set of observations. However, like the moving average model, it also has several disadvantages. In particular this model does not produce an actual equation. Therefore, it is not all that

useful as a medium-long range forecasting tool. It can only reliably be used to forecast a few periods into the future.

- **9: Naive forecast:** A naive forecasting model is a special case of the moving average forecasting model where the number of periods used for smoothing is 1. Therefore, the forecast for a period, t, is simply the observed value for the previous period, t-1.

Due to the simplistic nature of the naive forecasting model, it can only be used to forecast up to one period in the future. It is not at all useful as a medium-long range forecasting tool.

Some models need a periodicy to defined which can be passed with the optional argument 5. The periodicy defines the number of values for a single period – for instance if you want to forecast months in a year, the periodicy should be set to 12.

The optional last arguments allows to define an iteration for the forecast in order to create missing values in the past by forecasting them and then use these values as a base for further iterations.

Examples

```
FORECAST( Amount(), Time, LEVEL( Time, 2 ) )
```

Best forecast for the fact Amount() for the current Time based on the whole level 2 of the time-period.

```
FORECAST( Amount(), Time, LEVEL( Time, 2 ), 8, 12 )
```

Weighted moving average forecast for the current time, based on the whole level 2 of the time-period with a periodicy of 12 (number of months in a year).

See also

REGRESSION, SPLINE

FPOP

Syntax

```
FPOP( expression: <Any> )
```

Since

2.2

Return-type

Any (equal to the argument-type)

Description

Many functions influence the current filter, i.e. when iterating the input-set for a MATCH or SORT function. Furthermore, the iterations of headers, selector-options change the filter.

With this function, you can access the previous filter from the filter-stack before it was influenced by a function or iteration. The expression passed as argument will be simply executed with the previous version of the filter.

In addition, you can nest this function to access even earlier versions of the filter.

Examples

```
FPOP( Product )
```

Returns the selection of the Product-dimension before it was changed by the last function or iteration.

```
FPOP( COUNT( Product ) )
```

Counts the number of originally selected products before the last iteration.

```
FPOP( FPOP( Product ) )
```

Climbs up two levels in the filter-stack.

See also

FPUSH

FPUSH

Syntax

```
FPUSH( keys: <Key>, expression: <Any> )
```

Since

2.2

Return-type

Any (equal to the argument-type)

Description

The function FPUSH allows to execute an expression with a different filter than the current.

The expression passed as second argument will be executed after the first argument was applied to the filter. If the a is NULL, the filter will remain unchanged. This function does alter not the filter of the outer scope.

Examples

```
FPUSH( PREV( Product ), Amount() )
```

= Amount(PREV(Product))

```
FPUSH( PREV( Product ), Amount() / Quantity() )
```

= Amount(PREV(Product)) / Quantity(PREV(Product))

```
FPUSH( NULL, Amount() )
```

= Amount()

See also

FPOP

GETDAY

Syntax

```
<getday-expression> := 'GETDAY(' date: <Date> ')'
```

Since

2.5

Return-type

Integer

Description

Returns the day of the month of the passed date. If the argument is NULL, the function also returns NULL.

Examples

```
GETDAY( TODATE( '2011-03-31', 'yyyy-MM-dd' ) )
```

Returns 31 (the day of month of the argument).

See Also

GETDAYOFWEEK, GETHOUR, GETMILLISECOND, GETMINUTE, GETMONTH, GETSECOND, GETYEAR

GETDAYOFWEEK

Syntax

```
<getdayofweek-expression> := 'DAYOFWEEK(' date: <Date> ')'
```

Since

3.1

Return-type

Integer

Description

The GETDAYOFWEEK function returns the day of the week of the argument. The result is an integer number representing the different days:

Result	Day of week
1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday
7	Saturday

If any of the arguments is NULL, the function returns also NULL.

Examples

```
DAYOFWEEK( Project.End, Project.Start ) / ( 24 * 60 * 60 *
1000 )
```

Returns 0 (for “Sunday”).

See also

GETDAY, GETHOUR, GETMILLISECOND, GETMINUTE, GETMONTH, GETSECOND, GETYEAR

GETHOUR

Syntax

```
<gethour-expression> := 'GETHOUR(' timestamp: <Date> ')'
```

Since

2.5

Return-type

Integer

Description

Returns the hour of the day of the passed date. If the argument is NULL, the function also returns NULL.

Examples

```
GETHOUR( TODATE( '21:29:14 521', 'HH-mm-ss SSS' ) )
```

Returns 21 (the hour of the argument).

See Also

GETDAY, GETDAYOFWEEK, GETMILLISECOND, GETMINUTE, GETMONTH, GETSECOND, GETYEAR

GETMINUTE

Syntax

```
<getminute-expression> := 'GETMINUTE(' timestamp: <Date> ')'
```

Since

2.5

Return-type

Integer

Description

Returns the minute of the passed date. If the argument is NULL, the function also returns NULL.

Examples

```
GETDAY( TODATE( '21:29:14 521', 'HH-mm-ss SSS' ) )
```

Returns 29 (the minute of the argument).

See Also

GETDAY, GETDAYOFWEEK, GETHOUR, GETMILLISECOND, GETMONTH, GETSECOND, GETYEAR

GETMILLISECOND

Syntax

```
<getmillisecond-expression> := 'GETMILLISECOND(' timestamp:
<Date> ')'
```

Since

2.5

Return-type

Integer

Description

GETMILLISECOND returns the milliseconds of the passed date. If the argument is NULL, the function also returns NULL.

Examples

```
GETDAY( TODATE( '21:29:14 521', 'HH-mm-ss SSS' ) )
```

Returns 521 (the milliseonds of the argument).

See Also

GETDAY, GETDAYOFWEEK, GETHOUR, GETMINUTE, GETMONTH, GETSECOND, GETYEAR

GETMONTH

Syntax

```
<getmonth-expression> := 'GETMONTH(' date: <Date> ')'
```

Since

2.5

Return-type

Integer

Description

Returns the month of the passed date. If the argument is NULL, the function also returns NULL.

Examples

```
GETDAY( TODATE( '2011-03-31', 'yyyy-MM-dd' ) )
```

Returns 3 (the month of the argument).

See Also

GETDAY, GETDAYOFWEEK, GETHOUR, GETMILLISECOND, GETMINUTE, GETSECOND, GETYEAR

GETSECOND

Syntax

```
<getsecond-expression> := 'GETSECOND(' timestamp: <Date> ')'
```

Since

2.5

Return-type

Integer

Description

Returns the second of the passed date. If the argument is NULL, the function also returns NULL.

Examples

```
GETDAY( TODATE( '21:29:14 521', 'HH-mm-ss SSS' ) )
```

Returns 14 (the second of the argument).

See Also

GETDAY, GETDAYOFWEEK, GETHOUR, GETMILLISECOND, GETMINUTE, GETMONTH, GETYEAR

GETYEAR

Syntax

```
<getyear-expression> := 'GETYEAR(' date: <Date> ')'
```

Since

2.5

Return-type

Integer

Description

GETDAY returns the year of the passed date. If the argument is NULL, the function also returns NULL.

Examples

```
GETDAY( TODATE( '2011-03-31', 'yyyy-MM-dd' ) )
```

Returns 2011 (the year of the argument).

See Also

GETDAY, GETDAYOFWEEK, GETHOUR, GETMILLISECOND, GETMINUTE, GETMONTH, GETSECOND

GREATER

Syntax

```
<greater-expression> := 'GREATER(' values1: <Any> ',' values2: <Any> ')'

<greater-expression> := <Any> '>' <Any>
```

Since

1.0

Return-type

Boolean

Description

This function tests if each value of the first argument is greater than the corresponding value of the second argument. If both arguments have a different size, the function returns FALSE. If one of the arguments is NULL, the function returns NULL.

Instead of this function, you also can use the operator ">".

Examples

```
GREATER( 2, 1 )
```

= TRUE

```
2 > 1
```

= TRUE

```
GREATER( 2, 2 )
```

= FALSE

```
2 > 2
```

= FALSE

```
GREATER( NULL, 2 )
```

= NULL

```
GREATER( JOIN( 2, 3 ), JOIN( 1, 2 ) )
```

= TRUE

See also

EQUAL, GREATER_OR_EQUAL, LESS, LESS_OR_EQUAL, UNEQUAL

GREATER_OR_EQUAL

Syntax

```
<greaterorequal-expression> := 'GREATER_OR_EQUAL(' values1:
<Any> ',' values2: <Any> ')'

<greaterorequal-expression> := <Any> '>=' <Any>
```

Since

1.0

Return-type

Boolean

Description

This function tests if each value of the first argument is greater or equal than the corresponding value of the second argument. If both arguments have a different size, the function returns FALSE. If one of the arguments is NULL, the function returns NULL.

Instead of this function, you also can use the operator ">=".

Examples

```
GREATER_OR_EQUAL( 2, 1 )
```

= TRUE

```
2 >= 1
```

= TRUE

```
GREATER_OR_EQUAL( 2, 2 )
```

= TRUE

```
2 >= 2
```

= TRUE

```
GREATER_OR_EQUAL( NULL, 2 )
```

= NULL

```
GREATER_OR_EQUAL( JOIN( 2, 2 ), JOIN( 1, 2 ) )
```

= TRUE

See also

EQUAL, GREATER, LESS, LESS_OR_EQUAL, UNEQUAL

HASACCESS

Syntax

```
<hasaccess-expression> := 'HASACCESS(' keys: <Key> ')'
```

Since

2.2.0

Return-type

Boolean

Description

This function checks if the current user has access to the keys passed as the argument. This function can i.e. be used for the definition of access rules inside the configuration editor.

If the argument is NULL, then the function returns also NULL.

Examples

```
HASACCESS( Product.Manufacturer )
```

In an access rule for the Product dimension, this would grant access to a user for all products of the manufacturers he already has access to.

HASCHILDREN

Syntax

```
<haschildren-expression> := 'HASCHILDREN(' keys: <Key> ')'
```

Since

2.2.2

Return-type

Boolean

Description

This function checks if any of the keys, passed as argument, has at least one child. If the argument contains more than one key, multiple values are returned.

If NULLis passed as argument, the function returns NULL.

Examples

```
HASCHILDREN( Time:'Jan/2011' )
```

= TRUE

```
HASCHILDREN( Time:'01.01.2011' )
```

= FALSE

```
HASCHILDREN( Time:'Jan/2011' | Time:'01.01.2011' )
```

= TRUE | FALSE

```
HASCHILDREN( NULL )
```

= NULL

See also

CHILDREN, ISLEAF

HASKEYS

Syntax

```
<haskeys-expression> := 'HASKEYS(' keys: <Key> { ',' keys:
<Key> } ')'
```

Since

1.1

Return-type

Boolean

Description

The function HASKEYS checks if the current selection (filter) contains at least one key from each argument (usually each argument is only one key). This is very useful for building match-expressions, for instance, in cubes or formulas.

If any of the arguments is NULL, the function returns NULL.

Examples

```
HASKEYS( Fact:Amount )
```

Does the current filter contain the fact "Amount"?

```
HASKEYS( Fact:Amount, Time:'Jan/2011' )
```

Does the current filter contain the fact "Amount" **and** January?

```
HASKEYS( Time:'Jan/2011' | Time:'Feb/2011' )
```

Does the current filter contain the January **or** February?

See also

HASLEVEL, HASPOSITION

HASLEVEL

Syntax

```
<haslevel-expression> := 'HASLEVEL(' keys: <Key> { ',' levels:
<Integer> } ')'
```

Since

1.1

Return-type

Boolean

Description

This function checks if any of the keys of argument 1 has the level defined by the second argument.

If any value is NULL, the function returns NULL.

Examples

```
HASLEVEL( Time, 1, 2 )
```

Is a time-key of level 1 or 2 selected?

See also

HASKEYS, HASPOSITION

HASPOSITION

Syntax

```
<hasposition-expression> := 'HASPOSITION(' keys: <Key> { ','
position: <Integer> } ')'
```

Since

2.0

Return-type

Boolean

Description

This function checks if the keys passed as argument has the position defined in the second argument. The position of a key is its position in its parent child-list, beginning with 0.

If any argument is NULL, the function returns NULL.

Examples

```
HASPOSITION( Time:'Jan/2011', 0 )
```

= TRUE

```
HASPOSITION( Time:'Jan/2011', 1 )
```

= FALSE

See also

HASKEYS, HASLEVEL, HASPOSITION

HASROLES

Syntax

```
<hasroles-expression> := 'HASROLES(' role-name: <String> { ',' role-name: <String> } ')'
```

Since

2.0

Return-type

Boolean

Description

This function checks if the current user has all roles passed as arguments.

If any argument has more than one value, the user only needs to have one of the roles defines in this argument to pass the check.

If any of the arguments is NULL, the function returns NULL.

Examples

```
HASROLES( 'iolapAdmin', 'iolapUser' )
```

Is the current user admin and user?

```
HASROLES( 'iolapAdmin' | 'iolapUser' )
```

Is the current user admin or user?

See also

HASUSER, USER

HASUSER

Syntax

```
<hasuser-expression> := 'HASUSER(' user-name: <String> { ',' user-name: <String> } ')'
```

Since

2.0

Return-type

Boolean

Description

This function checks if the current user name matches one of the names passed as arguments.

If any of the arguments is NULL, the function returns NULL.

Examples

```
HASUSER( 'guest', 'admin' )
```

Is the current user named 'guest' or 'admin'?

See also

HASROLES, USER

HIERARCHIZE

Syntax

```
<hierarchize-expression> := 'HIERARCHIZE(' keys: <Key> ')'
```

Since

2.7

Return-type

Key

Description

This function brings all keys of the argument into their natural order: All keys are sorted behind their parents, and all keys with the same parent are sorted by their natural order inside the dimension.

NULL values are removed from the result.

Examples

```
HIERARCHIZE( Product:Notebook1, Product:Notebooks,
Product:Notebook2 )
```

= Product:Notebooks, Product:Notebook1, Product:Notebook2

```
HIERARCHIZE( Product:Notebook1, NULL, Product:Notebooks,
Product:Notebook2 )
```

= Product:Notebooks, Product:Notebook1, Product:Notebook2

See also

SORT

IIF

Syntax

```
<iif-expression> := 'IIF(' condition: <Boolean> ',' result-
for-true: <Any> ',' result-for-false: <Any> ')'
```

Since

1.2

Return-type

The return-type of the second and third argument (the super-type of them).

Description

The IIF function allows the conditional evaluation of two arguments: Depending of the first argument's result (which is a boolean expression), the function evaluates the second or third argument and returns its result or NULL:

- If argument 1 results to TRUE, the second argument is evaluated and returned.

- If argument 1 results to FALSE, the third argument is evaluated and returned.
- If argument 1 results to NULL, the result is NULL.

The IIF function can be nested to check more than one condition, but for more than on condition, we recommend using the CASE or SWITCH function.

Examples

```
IIF(
  HASLEVEL( Time, 1 ), 'Year',
  IIF( HASLEVEL( Time, 2 ), 'Month', 'Day' )
)
```

Returns 'Year', 'Month' or 'Day', depending on the level of the selected time.

See also

CASE, SWITCH

IN

Syntax

```
<in-expression> := 'IN(' search-values: <Any> ',' values:
<Any> ')'

<in-expression> := <Any> 'IN' <Any>
```

Since

1.2

Return-type

Boolean

Description

This function checks whether all values from the first argument are contained in the second argument.

Instead of this function, you also can use the operator "IN".

Examples

```
IN( Article.Color, 'Red' | 'Green' )
```

Is the current article red or green?

```
Article.Color IN ( 'Red' | 'Green' )
```

Same as above

See also

CONTAINS

INTERSECT

Syntax

```
<intersect-expression> := 'INTERSECT(' values: <Any> { ',' values: <Any> } ')'
```

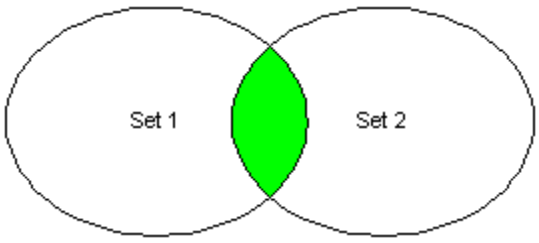
Since

1.2

Return-type

Supertype of all argument-types.

Description



The function INTERSECT computes the intersection of all arguments (the values contained in all arguments). The return-type is the most common super type of all arguments.

Examples

```
INTERSECT( Article.Customer, Region.Customer )
```

Returns the customers of the current article AND region

```
INTERSECT( ( 10 | 20 | 30 ), ( 20 | 30 | 40 ) )
```

= 20 | 30

See also

JOIN, WITHOUT

ISCHILDOF

Syntax

```
<ischildof-expression> := 'ISCHILDOF(' child: <Key> ',' <Key>
{ ',' parent: <Key> } ')'
```

Since

1.2

Return-type

Boolean

Description

This function checks whether the key from the first argument is equal to or children of at least one of the keys from the other arguments.

A key A is a child of a key B, if:

- B is the direct parent of A,
- or the parent of A is a child of B.

If more than one key is passed as first argument, the function checks all keys and returns multiple values.

If one of the arguments is NULL, the function returns NULL.

Examples

```
ISCHILDOF( Time:'Jan/2011', Time:'2011' )
```

= TRUE

```
ISCHILDOF( Time:'Jan/2010', Time:'2011', Time:'2010' )
```

= TRUE

```
ISCHILDOF( Time:'2011', Time:'Jan/2011' )
```

= FALSE

```
ISCHILDOF( Time:'2011', NULL )
```

= NULL

```
ISCHILDOF( NULL, Time:'Jan/2011' )
```

= NULL

See also

ISPARENTOF

ISEMPTY

Syntax

```
<isempty-expression> := 'ISEMPTY(' values: <Any> ')'
```

Since

2.5

Return-type

Boolean

Description

This function checks if the argument is an empty list and contains no elements. NULLvalues are treated like all other elements.

This function is equal to "COUNT(<Any>) = 0".

Examples

```
ISEMPTY( EMPTY() )
```

= TRUE

```
ISEMPTY( 10 )
```

= FALSE

```
ISEMPTY( NULL )
```

= FALSE

See also

ISNULL, EXISTS

ISLEAF

Syntax

```
<isleaf-expression> := 'ISLEAF( keys: <Key> )'
```

Since

2.6.1

Return-type

Boolean

Description

Returns TRUE if the key passed as argument is a leaf and has no children. If more than one key is passed as argument, the function returns the same number of boolean values as the result.

If the argument is NULL, the function returns NULL.

Examples

```
ISLEAF( Product:'All products' )
```

= FALSE

```
ISLEAF( Product:'All products' | NULL )
```

= FALSE | NULL

See also

HASCHILDREN

ISNULL

Syntax

```
<isnull-expression> := 'ISNULL(' values: <Any> ')'
```

Since

1.2

Return-type

Boolean

Description

This function checks if the argument is empty or contains any NULL.

Since version 2.7 it also checks for empty arguments, because fact-functions return empty results instead of NULL.

Examples

```
ISNULL( NULL )
```

= TRUE

```
ISNULL( 10 )
```

= FALSE

```
ISNULL( ( 10 | NULL ) )
```

= TRUE

```
ISNULL( EMPTY() )
```

= TRUE

See also

EXISTS

ISPARENTOF

Syntax

```
<isparentof-expression> := 'ISPARENTOF(' parent: <Key> ','
chil: <Key> ')'
```

Since

1.2

Return-type

Boolean

Description

Checks, whether all keys from the first argument are parents of all keys from the second argument. A key A is a father of a key B, if:

- B is directly placed under A,
- A child of A is the parent of B.

If one of the arguments is NULL, the function returns NULL. If the first argument is empty and contains no key, the function returns TRUE.

Examples

```
ISPARENTOF( Time:'2011', Time:'Jan/2011' )
```

= TRUE

```
ISPARENTOF( Time:'Jan/2011', Time:'2011' )
```

= FALSE

```
ISPARENTOF( Time:'2011', NULL )
```

= NULL

```
ISPARENTOF( NULL, Time:'2011' )
```

= NULL

See also

ISCHILDOF

ITERATIONKEY

Syntax

```
<iterationkey-expression> := 'ITERATIONKEY()'
```

Since

2.2.0

Return-type

Key

Description

If this function is used inside a header and if the header was created by a header-iteration, this function returns the single iteration-key for this header.

In a header without an iteration, the function returns NULL.

Examples

```
ITERATIONKEY()
```

I.e. returns 'Feb/2011' for one of the headers generated by the iteration 'Time::MONTH'.

See also

DRILLKEY, DRILLLEVEL

IVALUE

Syntax

```
<ivalue-expression> := 'IVALUE()'
```

Since

2.5.0

Return-type

Value

Description

This function returns the iterated value in the functions FOREACH, MATCH or SORT.

Examples

```
MATCH( PRODUCT.Name, STARTWITH( {IVALUE()}, 'P' ) )
```

This example return all product names beginning with 'P'.

See also

FOREACH, MATCH, SORT

JOIN

Syntax

```
<join-expression> := 'JOIN(' values: <Any> { ',' values: <Any> } ')'

<join-expression> := <Any> '|' <Any>
```

Since

1.2

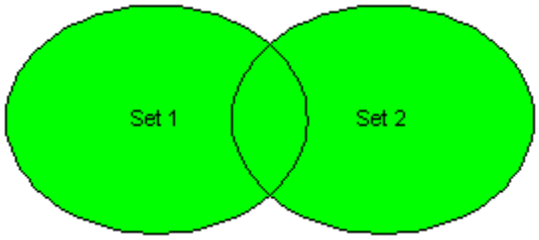
Return-type

Depending of both argument's type (their super-type).

Description

The function JOIN concatenates all results from all arguments to one joined result. The result of the first argument is added to result at first, then from the second and so on.

If any of the values is NULL, the NULL is simply added to result.



Instead of the function JOIN you also can use the operator "|".

Examples

```
JOIN( LEVEL( Time, 1 ), LEVEL( Time, 2 ) )
```

is equal to LEVEL(Time, 1, 2)

```
JOIN( NULL )
```

= NULL

See also

DISTINCT, INTERSECT, MIX, WITHOUT

KEYINDEX

Syntax

```
<keyindex-expression> := 'KEYINDEX( keys: <Key> )'
```

Since

2.6.0

Return-type

Integer

Description

Returns the internal index of the key(s). Each key has a unique index within its dimension. By default, the root key of a dimension has always the index 0. All other keys have numbers greater than zero.

If the argument is NULL, the function returns NULL.

Examples

```
KEYINDEX( Product:'All products')
```

= 0

See also

POSITIONOF

LANGUAGE

Syntax

```
<language-expression> := 'LANGUAGE()'
```

Since

2.2.0

Return-type

String

Description

Returns the language-code of the session of the current user. The language and country are usually passed by the Browser.

Examples

LANGUAGE()

= I.e. 'de'

See also

COUNTRY, LOCALE

LAST

Syntax

```
<last-expression> := 'LAST(' list: <Any> [ ',' size: <Integer>
] ')'
```

Since

1.2

Return-type

The type of the first argument.

Description

This function returns the last N values of the first argument. If no size N is defined (the second argument), the function only returns the last element of argument.

If one of the arguments is NULL, the function returns NULL.

Examples

```
LAST( LEVEL( Time, 2 ) )
```

= Time:'Dec/20011'

```
LAST( LEVEL( Time, 2 ), 3 )
```

= Time:'Oct/2011' + Time:'Nov/2011' + Time:'Dec/2011'

```
LAST( NULL )
```

= NULL

See also

FIRST

LEAFS

Syntax

```
<leafs-expression> := 'LEAFS(' keys: <Any> ')'
```

Since

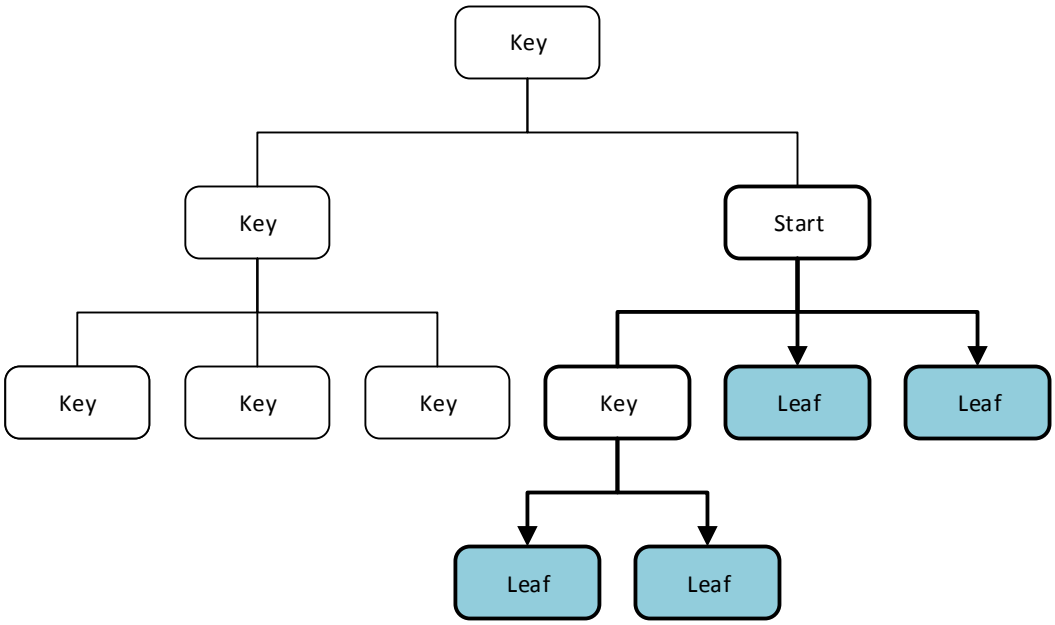
1.1

Return-type

Key

Description

Returns all leafs (keys without children) of the arguments' family. The leafs don't need to be direct children of the argument and can be further down in the hierarchy.



If the argument is NULL, the function returns NULL. If the argument only contains leafs, the function returns an empty result.

Examples

```
LEAFS( Time:'2011' )
```

I.e. returns Time:'Jan/2011' + ... + Time:'Dec/2011'

```
LEAFS( NULL )
```

= NULL

See also

CHILDREN, FAMILY, NONLEAFS

LEFT

Syntax

```
<left-expression> := 'LEFT(' text: <String> ',' size:
<Integer> ')'
```

Since

2.0

Return-type

String

Description

The function LEFT returns the first N (defined by the second argument) characters of the string passed as first argument. If the length of the string is smaller than N, the original string is returned.

If any argument is NULL, the function returns NULL.

Examples

```
LEFT( 'Hello world', 5 )
```

= 'Hello'

```
LEFT( 'Hello world', 50 )
```

= 'Hello world'

```
LEFT( NULL, 5 )
```

= NULL

```
LEFT( 'Hello world', NULL )
```

= NULL

See also

LIMIT, RIGHT, SUBSTR, SUBSTRINGBEHIND

LESS

Syntax

```
<less-expression> := 'LESS(' values1: <Any> ',' values2: <Any>
')'

<less-expression> := <Any> '<' <Any>
```

Since

1.0

Return-type

Boolean

Description

This function tests if each value of the first argument is greater than the corresponding value of the second argument. If both arguments have a

different size, the function returns FALSE. If one of the arguments is NULL, the function returns NULL.

Instead of this function, you also can use the operator "<".

Examples

```
LESS( 1, 2 )
```

= TRUE

```
1 < 2
```

= TRUE

```
LESS( 2, 2 )
```

= FALSE

```
2 < 2
```

= FALSE

```
LESS( NULL, 2 )
```

= NULL

```
LESS( 2 | 3, 1 | 2 )
```

= FALSE

See also

EQUAL, GREATER, GREATER_OR_EQUAL, LESS_OR_EQUAL, UNEQUAL

LESS_OR_EQUAL

Syntax

```
<lessorequal-expression> := 'LESSOREQUAL(' values1: <Any> ',' values2: <Any> ')'

<lessorequal-expression> := <Any> '<=' <Any>
```

Since

1.0

Return-type

Boolean

Description

This function tests if each value of the first argument is less or equal to the corresponding value of the second argument. If both arguments have a different size, the function returns FALSE.

If one of the arguments is NULL, the function returns NULL.

Instead of this function, you also can use the operator "<=".

Examples

```
LESS_OR_EQUAL( 1, 2 )
```

= TRUE

```
1 <= 2
```

= TRUE

```
LESS_OR_EQUAL( 2, 2 )
```

= TRUE

```
2 <= 2
```

= TRUE

```
LESS_OR_EQUAL( NULL, 2 )
```

= NULL

```
LESS_OR_EQUAL( 2 | 3, 2 | 2 )
```

= TRUE

See also

EQUAL, GREATER, GREATER_OR_EQUAL, LESS, UNEQUAL

LEVEL

Syntax

```
<level-expression> := 'LEVEL(' dimension: <Key> { ',' level-
number: <Integer> } ')'
```

Since

1.1

Return-type

Key

Description

The function returns one or more complete levels from the hierarchy of the dimension indicated with argument 1. The levels are defined by the optional arguments 2 - n. If the first argument is NULL or any level is NULL, the result is NULL. If no level is defined, an empty result is returned.

Note: This function is not affected by the current filter. It will always return complete and unfiltered levels.

Examples

```
LEVEL( Time, 0 )
```

= Time:'Complete Period'

```
LEVEL( Time, 1, 2 )
```

= Time:'2011' + Time:'Jan/2011' + ... + Time:'Dec/2011'

```
LEVEL( Time )
```

= Empty result

```
LEVEL( NULL )
```

= NULL

```
LEVEL( Time, NULL )
```

= NULL

See also

ALL, LEVELOF

LEVELNAMES

Syntax

```
<levelnames-expression> := 'LEVELNAMES(' dimension: <Key> ')'
```

Since

2.1

Return-type

String

Description

This function returns the names of all hierarchy-levels of the dimension indicated by the argument. The order of the names is top down.

If the argument is NULL, the function returns NULL.

Examples

```
LEVELNAMES( Time )
```

I.e. returns 'ROOT' | 'YEAR' | 'DAY'

See also

DIMENSIONNAME, DIMENSIONNAMES

LEVELOF

Syntax

```
<levelof-expression> := 'LEVELOF(' keys: <Key> ')'
```

Since

2.0

Return-type

Integer

Description

Returns the hierarchy level number of the keys defined by the argument. The most upper level (which only the root-key belongs to) has the level 0. All deeper levels have higher numbers.

If the argument contains multiple keys, the function returns the same number of values.

If the argument is NULL, the function returns NULL.

Examples

```
LEVELOF( Time:'Complete Time' )
```

= 0

```
LEVELOF( Time:'2011' )
```

= 1

```
LEVELOF( Time:'Jan/2011' )
```

= 2

```
LEVELOF( Time:'01.01.2011' )
```

= 3

```
LEVELOF( Time:'Jan/2011' | Time:'01.01.2011' )
```

= 2 | 3

See also

HASLEVEL, POSITIONOF

LIKE

Syntax

```
<like-expression> := 'LIKE(' text: <String> ',' pattern:
<String> ')'

<like-expression> := <String> 'LIKE' <String>
```

Since

2.5

Return-type

Boolean

Description

The LIKE function compares the string passed as first argument with the pattern passed as second argument. It returns TRUE if the string matches the pattern or FALSE if not. The pattern can use the wild cards '?' for a single and '*' for multiple characters.

If any of the arguments is NULL, the function returns NULL.

Instead of this function, you also can use the operator "LIKE".

Examples

```
LIKE( 'Hello world', '*world' )
```

= TRUE

```
'Hello world' LIKE '*world'
```

= TRUE

```
LIKE( 'Hello world', 'H?llo world' )
```

= TRUE

```
LIKE( 'Hello world', '*ship' )
```

= FALSE

```
LIKE( NULL, '*world' )
```

= NULL

See also

EQUAL, REGEXP

LIMIT

Syntax

```
<limit-expression> := 'LIMIT(' text: <String> [ ',' size:
<Integer> [ ',' postfix: <String> ] ] ')'
```

Since

1.2

Return-type

String

Description

This function limits the size of the strings passed as arguments:

- If their size is greater than the maximum-size defined by argument 2, the strings will be cut to this size and the chars "..." will be appended.
- Otherwise the original string is returned.

If the second argument is left out, the maximum-size will be the default value 50. With the third and optional argument, you can change the appendix "..." to any other string-pattern.

If any argument is NULL, the function returns NULL.

Examples

```
LIMIT( 'Hello World', 8 )
```

= 'Hello...'

```
LIMIT( 'Hello World', 8, '!' )
```

= 'Hello!'

```
LIMIT( NULL, 8 )
```

= NULL

See also

BEAUTIFY, LEFT, SUBSTR

LOCALE

Syntax

```
<locale-expression> := 'LOCALE()'
```

Since

2.2.0

Return-type

String

Description

Returns the locale code of current user. The locale contains both the country- and language-code and is passed by the users' Browser.

Examples

```
LOCALE()
```

= 'DE_de'

See also

COUNTRY, LANGUAGE

LOG

Syntax

```
<log-expression> := 'LOG( <Number> )'
```

Since

2.7.0

Return-type

Double

Description

Returns the natural logarithm (base e) of the argument. Special cases:

- If the argument is NULLor less than zero, then the result is NULL.
- If the argument is positive infinity, then the result is also positive infinity.
- If the argument is positive zero or negative zero, then the result is negative infinity.

Examples

```
LOG(100)
```

= 4.605170185988092

```
LOG(-100)
```

= NULL

See also

EXP, POW

LOOKUP

Syntax

```
<lookup-expression> := 'LOOKUP(' keys: <Key> { ',' search-
filter: <Key> } ')'
```

Since

2.0

Return-type

Key

Description

This function evaluates, by immediately querying the cubes, for which keys of the first argument a fact (usually defined by the second argument) is available and not NULL.

When querying the cubes, the current filter plus all following arguments is considered. I.e. the example “LOOKUP(PRODUCT, Amount:Quantiy' “), with a filter “Time:2011 | Fact:Amount”, searches all products with for the year 2011 and the fact “Quantity” (the year is taken from the original filter, but the fact is overridden by the second argument).

The LOOKUP-function is very important for building reports showing values for sparse filled cubes. The “Suppress Rows” and “Suppress Columns” properties o the query have the same effect but can be much slower because the rows and columns are removed from the result after the it was generated. With the LOOKUP function, tables can be reduced before the table is generated. Furthermore, the LOOKUP function can be used for other elements, i.e. in Selector options.

If any of the arguments is NULL, the function returns NULL.

Examples

```
LOOKUP( PRODUCT, Fact:Quantity )
```

Returns all products with value for the fact “Quantity” in the current-year, etc.

See also

SORT

LTRIM

Syntax

```
<ltrim-expression> := 'LTRIM(' text: <String> ')'
```

Since

2.2.6

Return-type

String

Description

The function LTRIM removes all leading white-spaces from the string passed as argument.

If the argument is NULL the function also returns NULL.

Examples

```
LTRIM( ' Hello World ' )
```

= 'Hello World '

```
LTRIM( NULL )
```

= NULL

See also

RTRIM, TRIM

MATCH

Syntax

```
<match-expression> := 'MATCH(' values: <Value> ',' condition-
expression: <Boolean> ')'

<match-expression> := <Value> '?' <Boolean>
```

Since

1.2, changed in 2.5.0

Return-type

The return type of the first argument

Description

This function filters a set of values (defined by the first argument) with a boolean match expression. It only returns the subset of the values for which the second argument result to TRUE.

Like the functions SORT and FOREACH, this function creates a copy of the current filter for each key and applies the key to it. Then the match-expression is executed with the new filter. If the result is TRUE, the value will be added to the result.

Since version 2.5.0, the function accepts all expression types as first argument. If the argument does not return keys, you can use the function IVALUE to access the iterated value.

Instead of this function you can also use the operator "?".

Examples

```
MATCH( LEVEL( Product, 1 ), Product.ProductType = 'Type A' )
```

Returns all products with type 'Type A'

```
LEVEL( Product, 1 ) ? Product.ProductType = 'Type A'
```

Same as above

See also

FIND, FOREACH, IVALUE, SORT

MATRIX

Syntax

```
<matrix-expression> := 'MATRIX(' start-x: <Number> [ ',' 
start-y: <Number> [ ',' end-x: <Number> [ ',' end-y: <Number> 
] ] } ')'
```

Since

2.0

Return-type

Value

Description

The MATRIX-function allows to access the values of cells inside the current table or sub result. For instance, you can use this function to build SUM-rows or -columns inside your queries.

The function expects at least one argument: The x-position of the cell in the same row, which value you want to access. If you want to access the value of another row, then you can pass the row-number as the second argument.

If only one or two arguments are passed, this function returns exactly one value. With the third and fourth argument, you can define a whole area,

and multiple values are returned: The third argument defines the maximum column, the fourth the maximum row to read. E.g. if you read an area of 5 columns and 5 rows, 25 values are returned. Empty cells return NULL.

In connection with this function, the functions X and Y are also interesting. These functions can help you to find the current position of a cell and access cells in a relative position to it.

With the functions MAX_Y, MAX_Y, MIN_X and MIN_Y you can find the first or last row or column of a result.

The function XHEADER and YHEADER help you to find a specific row or column by its header text.

If any argument contains NULLvalues, the function will return NULL.

Examples

```
MATRIX( X() + 1 )
```

Returns the value right to the current cell

```
SUM( TO_NUMBER( MATRIX( X(), MIN_Y(), X(), Y - 1() ) ) )
```

Creates a summary row

```
MATRIX( X(), NULL )
```

= NULL

See also

MAX_X, MAX_Y, MAX_Y, MIN_X, MIN_Y, SCREENY, X, XHEADER, Y, YHEADER

MAX

Syntax

```
<max-expression> := 'MAX(' values: <Number> { ',' values:
<Number> } ')'
```

Since

1.0

Return-type

Double

Description

The MAX-function returns the greatest value of all values passed as argument. NULL values are not included into the comparison. If all arguments are NULL, the function returns NULL.

Examples

```
MAX( Amount( LEVEL( Time, 3 ) ) )
```

Returns the greatest Amount for all days

See also

ADD, AVG, MIN, SUM

MAXKEY

Syntax

```
<maxkey-expression> := 'MAXKEY(' keys: <Key> ',' expression:
<Value> ')'
```

Since

2.2.6

Return-type

Key

Description

The MAXKEY function returns the key of the first argument with the greatest result for the expression passed as the second argument. If any argument is NULL, the function returns NULL.

Examples

```
MAXKEY( PRODUCT, Amount() )
```

Returns the product with the greatest amount (for the current filter).

See also

AVGKEY, MINKEY

MAX_X

Syntax

```
<max_x-expression> := 'MAX_X()'
```

Since

2.0

Return-type

Integer

Description

When executed in a header or cell, this function returns the X-coordinate of the last column in the current sub result.

When used outside a header the function raises an error.

Examples

```
MATRIX( MAX_X() )
```

Returns the value of the last column in the current row

See also

MATRIX, MAX_Y, MIN_X, MIN_Y, SCREENY, X, XHEADER, Y, YHEADER

MAX_Y

Syntax

```
<max_y-expression> := 'MAX_Y()'
```

Since

2.0

Return-type

Integer

Description

When executed in a header or cell, this function returns the Y-coordinate of the last row in the current sub result.

When used outside a header the function raises an error.

Examples

```
MATRIX( X(), MAX_Y() )
```

Returns the value of the last row in the current column

See also

MATRIX, MAX_X, MIN_X, MIN_Y, SCREENY, X, XHEADER, Y, YHEADER

MEDIAN

Syntax

```
<median-expression> := 'MEDIAN(' value: <Number> ')'
```

Since

3.0

Return-type

Double

Description

This function calculates the median for all values if the argument. This is a shortcut for the PERCENTILE function with a 50% percentile.

Examples

```
MEDIAN( 10 | 20 | 30 )
```

Returns the median (20) of all values

See also

PERCENTILE

MIN

Syntax

```
<max-expression> := 'MIN(' value: <Number> { ',' value:
<Number> } ')'
```

Since

1.0

Return-type

Double

Description

The MIN-function returns the smallest value of all values of all arguments. NULL values are not included into the comparison, but if all arguments are NULL, the function returns NULL.

Examples

```
MIN( Amount( LEVEL( Time, 3 ) )
```

Returns the smallest Amount for all days

See also

ADD, AVG, MAX, SUM

MINKEY

Syntax

```
<minkey-expression> := 'MINKEY(' keys: <Key> ',' expression:
<Value> ')'
```

Since

2.2.6

Return-type

Key

Description

The MINKEY function returns the key of the first argument with the smallest result for the expression passed as the second argument. If any argument is NULL, the function returns NULL.

Examples

```
MINKEY( PRODUCT, Amount() )
```

Returns the product with the smallest amount (for the current filter).

See also

AVGKEY, MAXKEY

MIN_X

Syntax

```
<min_x-expression> := 'MIN_X()'
```

Since

2.0

Return-type

Integer

Description

When executed in a header or cell, this function returns the X-coordinate of the first non-header column in the current sub result.

When used outside a header the function raises an error.

Examples

```
MATRIX( MIN_X(), MIN_Y(), MAX_X(), MIN_Y() )
```

Returns all values of the first row below the table header

See also

MATRIX, MAX_X, MAX_Y, MIN_Y, SCREENY, X, XHEADER, Y, YHEADER

MIN_Y

Syntax

```
<min_y-expression> := 'MIN_Y()'
```

Since

2.0

Return-type

Integer

Description

When executed in a header or cell, this function returns the Y-coordinate of the first non-header row in the current sub result.

When used outside a header the function raises an error.

Examples

```
MATRIX( MIN_X(), MIN_Y(), MIN_X(), MAX_Y() )
```

Returns all values of the column right of the header

See also

MATRIX, MAX_X, MAX_Y, MIN_X, SCREENY, X, XHEADER, Y, YHEADER

MIX

Syntax

```
<mix-expression> := 'MIX(' values: <Any> { ',' values: <Any> }
')'
```

Since

2.5

Return-type

Common super type of all arguments.

Description

The MIX function mixes all lists of all arguments by returning all first elements of all arguments, then all seconds elements and so on.

If any argument has fewer elements than the other arguments, the function will continue adding the elements of the other arguments until all elements are returned.

This function treats NULLs like any other value and NULL values will be merged to the result list.

Examples

```
MIX( ( 'A' | 'B' | 'C' ), ( 'D' | 'E' | 'F' ) )
```

= 'A' | 'D' | 'B' | 'E' | 'C' | 'F'

```
MIX( ( 'A' | 'B' | 'C' ), ( 'D' ) )
```

= 'A' | 'D' | 'B' | 'C'

```
MIX( ( 'A' | 'B' | 'C' ), NULL )
```

= 'A' | NULL | 'B' | 'C'

See also

JOIN

MOD

Syntax

```
<mod-expression> := 'MOD(' value: <Number> ',' divisor:
<Number> ')'

<mod-expression> := <Number> '%' <Number>
```

Since

1.2

Return-type

Double

Description

This function computes the remainder of the division of argument one by argument two. If one of both arguments is NULL, the result is NULL.

Instead of this function, you also can use the operator "%".

Examples

```
MOD( 5, 2 )
```

= 1

```
94 % 10
```

= 4

```
MOD( NULL, 1 )
```

= NULL

```
10 % NULL
```

= NULL

See also

DIV, MUL

MUL

Syntax

```
<mul-expression> := 'MUL(' values: <Number> ',' values:
<Number> ')'

<mul-expression> := <Number> '*' <Number>
```

Since

1.0

Return-type

Double

Description

This function multiplies all values of the arguments.

If any value is NULL, the function returns NULL.

Instead of this function, you also can use the operator "*".

Examples

```
MUL( 10, 42 )
```

= 420

```
10 * 42
```

= 420

```
42 * NULL
```

= NULL

See also

ADD, DIV, MOD, SUM

NEIGHBOURS

Syntax

```
<neighbours-expression> := 'NEIGHBOURS(' keys: <Key> ')'
```

Since

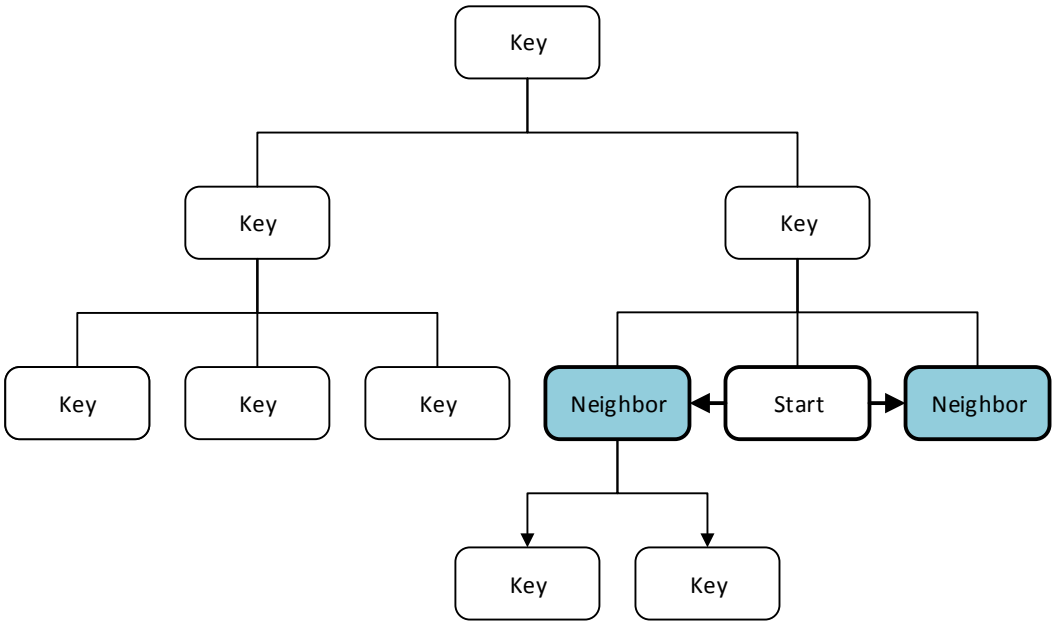
1.2

Return-type

Key

Description

This function determines all "neighbors" of the keys passed as argument. The "neighbors" of a key are all other keys with the same parent, without key itself.



If the argument is NULL, the function returns NULL.

Examples

```
NEIGHBOURS( Time:'May/2003' )
```

E.g. returns Time:'Apr/2003' + Time:'Jun/2003'

See also

CHILDREN, PARENT

NEXT

Syntax

```
<next-expression> := 'NEXT(' keys: <Key> [ ',' distance:
<Integer> [ ',' count: <Integer> ] ] ')'
```

Since

1.0

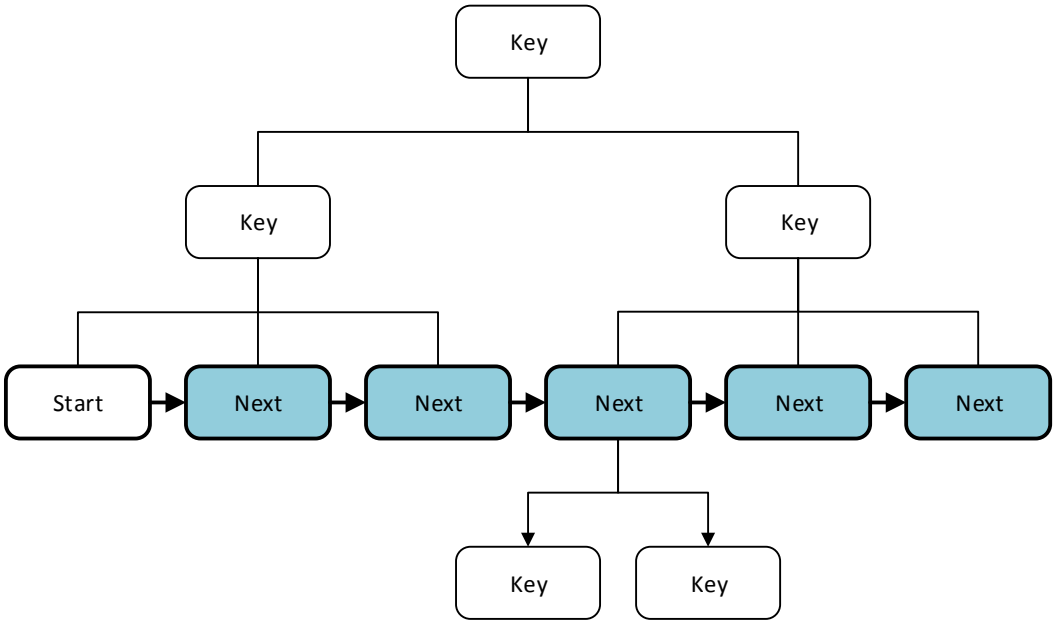
Return-type

Key

Description

Evaluates the following keys of the keys from the argument 1. The successor of a key is the next key within the same level of the dimension and must not have compellingly the same parent.

If no distance (argument 2) is defined, the directly following key is returned. Otherwise the key with the distance N from the key is returned. For instance, “NEXT(time, 2)” equals “NEXT(NEXT(time))”.



With the third argument, you can also define the number of keys you want the function to return. The default value is 1 and only a single key is returned.

If the argument is NULL, the result is NULL. If the keys have no successor, because they are last within the level, the result is empty.

Examples

```
NEXT( Time:'Jan/2011' )
```

= Time:'Feb/2011'

```
NEXT( Time:'Jan/2011', 3 )
```

= Time:'Apr/2011'

```
NEXT( Time:'Dec/2011 )
```

= Empty set

```
NEXT( Time:'Jan/2011', 1, 3 )
```

= Time:'Feb/2011' | Time:'Mar/2011' | Time:'Apr/2011'

```
NEXT( NULL )
```

= NULL

See also

PREV, PRED, SUCC, UPPERNEXT, UPPERPREV

NONFACTROOTS

Syntax

```
<nonfactroots-expression> := 'NONFACTROOTS()'
```

Since

1.2

Return-type

Key

Description

Returns the root-keys of all dimensions except the fact-dimension.

Examples

```
NONFACTROOTS()
```

I.e.. returns Time:'Complete Period' + ... + Product:'All Products'

See also

FACTROOT

NONLEAFS

Syntax

```
<nonleafs-expression> := 'NONLEAFS(' keys: <Key> ')'
```

Since

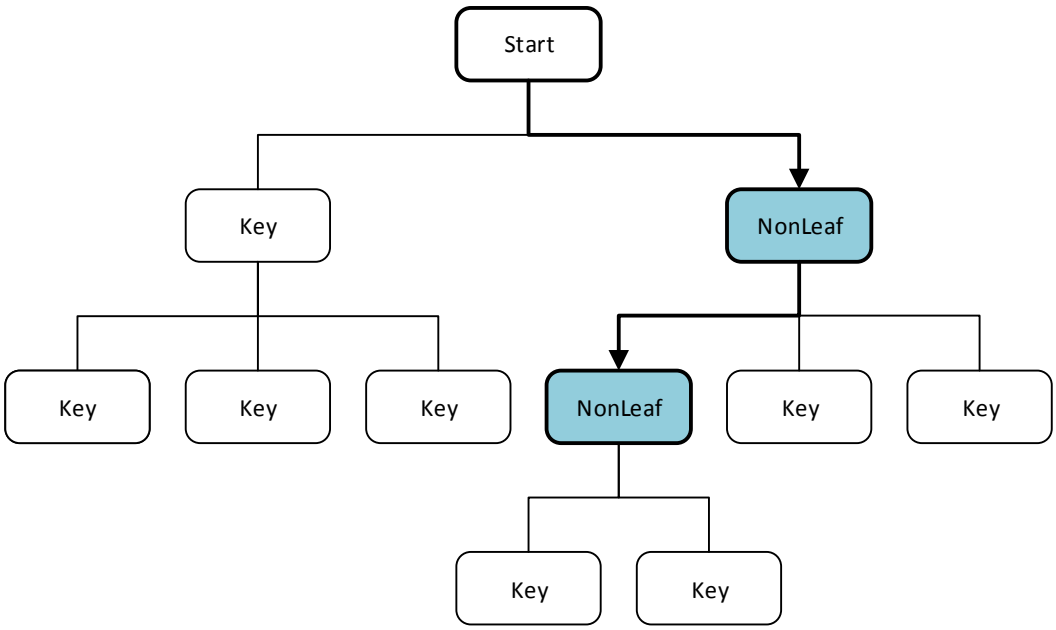
1.2

Return-type

Key

Description

The function NONLEAFS determines all keys of the arguments' family which also have own children.



If the argument is NULL, the function returns NULL.

Examples

```
NONLEAFS( Time:'2011' )
```

I.e.. returns Time:'Jan/2011', ..., Time:'Dec/2011' (but no days).

See also

ANCESTORS, CHILDREN, FAMILY, LEAFS, PARENT

NOT

Syntax

```
<not-expression> := 'NOT(' value: <Boolean> ')'
<not-expression> := '!' <Boolean> ')'
```

Since

1.0

Return-type

Boolean

Description

This logical function negates the boolean argument:

- If the argument is TRUE, the function returns FALSE.
- If the argument is FALSE, the function returns TRUE.
- If the argument is NULL, the function returns NULL.

Instead of this function, you can also use the operator “!”.

Examples

```
NOT( TRUE )
```

= FALSE

```
NOT( FALSE )
```

= TRUE

```
NOT( NULL )
```

= NULL

```
!TRUE
```

= FALSE

See also

AND, OR

NOW

Syntax

```
<now-expression> := 'NOW(' dimension: <Key> ',' pattern:
<String> ')'
```

Since

1.2

Return-type

Key

Description

The function NOW finds the key from a time dimension which corresponds to the current day, month, year, etc.

Because there is no special time dimension in instantOLAP, you can find the current time key of your time dimension with this function.

To find the current time key, you must pass the time dimension as the first argument. With the second argument, you must define a time-pattern, with which the desired key is searched.

Examples

```
NOW( Time, 'yyyy' )
```

= i.e. Time:'2011'

```
NOW( Time, 'MMM/yyyy' )
```

= i.e. Time:'Jan/2011'

```
NOW( Time, 'MMM/yy' )
```

= NULL

See also

FIND, TIMESTAMP, TODATE

NUMBER_RANGE

Syntax

```
<numberrange-expression> := 'NUMBER_RANGE(' start: <Integer>
',' end: <Integer> [ ',' step: <Integer> ] ')'
```

Since

3.1

Return-Type

Integer

Description

This function returns a series of Integers, beginning with the start value and ending with the end value. If the start value is higher than the end

value, the numbers in result will decrease from the higher to the lower value.

The optional step argument allows to define the steps with which the values will increase or decrease. Note that even if the start value is higher than the end value, this argument must be a positive value.

If any of the arguments is NULL, this function returns NULL.

Examples

```
NUMBER_RANGE( 10, 15 )
```

= 10 | 11 | 12 | 13 | 14 | 15

```
NUMBER_RANGE( 15, 10 )
```

= 15 | 14 | 13 | 12 | 11 | 10

```
NUMBER_RANGE( 10, 15, 2 )
```

= 10 | 12 | 14

```
NUMBER_RANGE( 15, 10, 2 )
```

= 15 | 13 | 11

```
NUMBER_RANGE( 10, NULL )
```

= NULL

See also

RANGE

OR

Syntax

```
<or-expression> := 'OR(' value1: <Boolean> ',' value2:
<Boolean> ')'

<or-expression> := <Boolean> 'OR' <Boolean>
```

Since

1.0

Return-type

Boolean

Description

The logical operator OR returns TRUE, FALSE or NULL, depending on the arguments' values:

- If any argument results to TRUE, the result is also TRUE.
- If both arguments result to FALSE, the result is also FALSE.
- If one argument results to NULL, the result is also NULL.

Instead of this function, you also can use the operator "OR".

Examples

```
OR( TRUE, TRUE )
```

= TRUE

```
OR( FALSE, TRUE )
```

= TRUE

```
OR( TRUE, FALSE )
```

= TRUE

```
OR( TRUE, NULL )
```

= TRUE

```
OR( FALSE, NULL )
```

= NULL

```
OR( NULL, NULL )
```

= NULL

See also

AND, OR

PARENT

Syntax

```
<parent-expression> := 'PARENT(' keys: <Key> ')'
```

Since

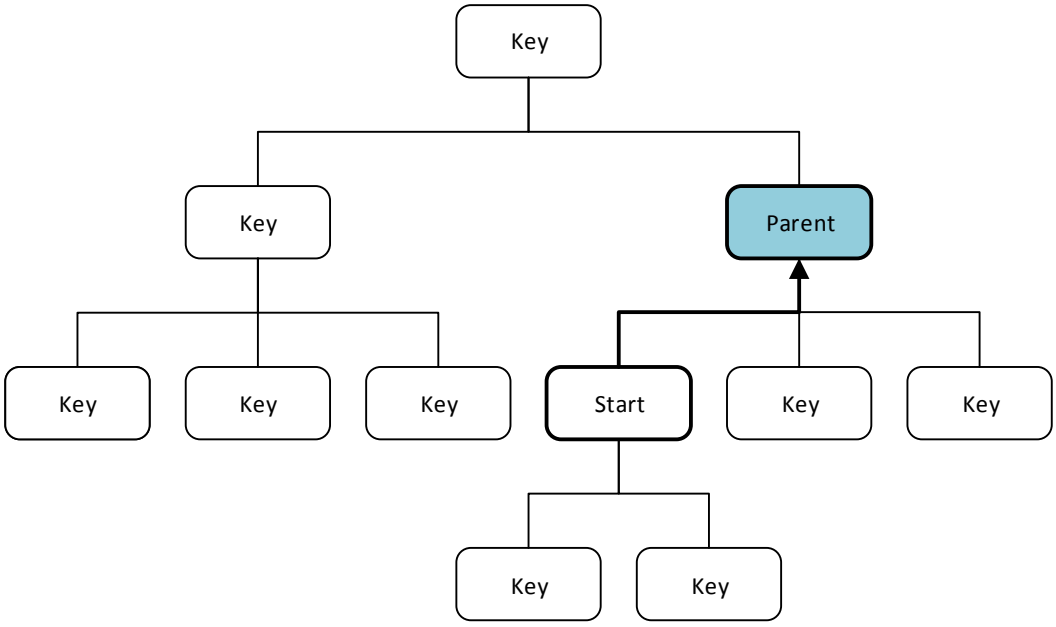
1.0

Return-type

Key

Description

The function PARENT returns the parent of a key. This is the key located directly above the argument in the hierarchy. All keys except the root-key have parents.



If the argument is NULL, this function returns NULL.

Examples

```
PARENT( Time:'Jan/2011' )
```

I.e.. returns Time:'Q1/2011'

```
PARENT( Time:'Complete Period' )
```

Returns NULL (given that Time:'Complete Period' is the root-key)

```
PARENT( NULL )
```

= NULL

See also

ANCESTORS, CHILDREN, FAMILY, ISPARENTOF, PEDIGREE

PEDIGREE

Syntax

```
<pedigree-expression> := 'PEDIGREE(' keys: <Key> ')'
```

Since

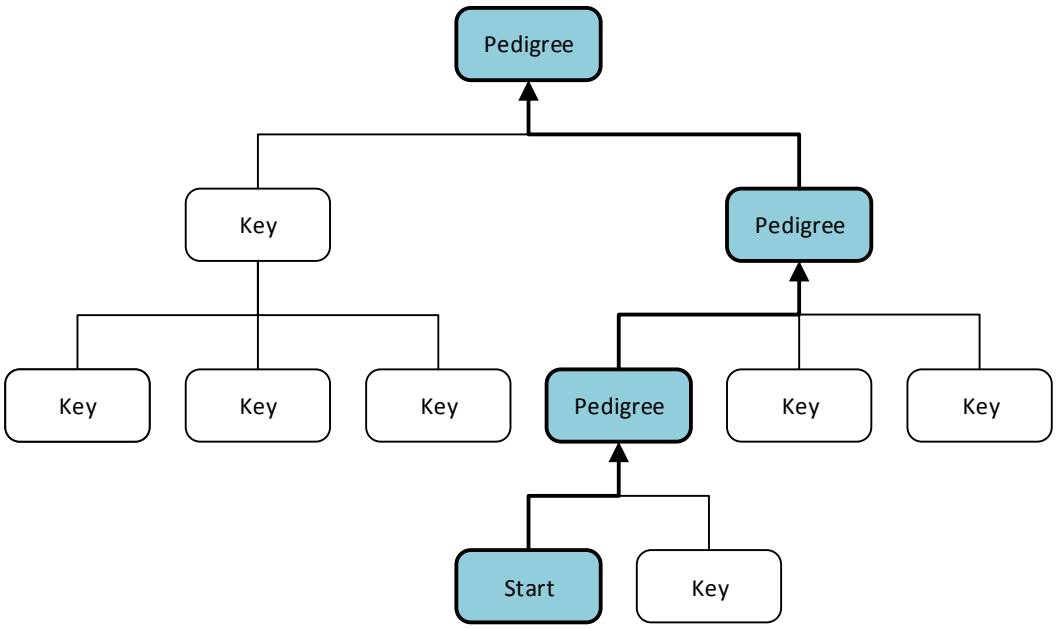
2.1.2

Return-type

Key

Description

Returns the keys passed as arguments together with all their ancestors. This function returns a distinct and ordered list - even if multiple keys from the argument have the same ancestors, they will only appear once in the result.



If the argument is NULL, the function returns NULL.

Examples

```
ANCESTORS( Time:'Jun/2011', Time:'May/2011' )
```

I.e. returns Time:'Complete Period' | Time:'2011' | Time:'Jun/2011' | Time:'May:2011

See also

ANCESTORS, PARENT

PERCENTILE

Syntax

```
<percentile-expression> := 'PERCENTILE(' values: <Number> ',' percentile: <Number> ')
```

Since

2.2.6

Return-type

Integer

Description

The function PERCENTILE calculates the percentile (defined by the second argument as a value between 0 and 100) for the values passed as the first argument.

Examples

```
PERCENTILE( 10 | 15 | 21 | 18, 90 )
```

= 21

See also

DEVIATION, MEDIAN, VARIANCE

PERMUTATE

Syntax

```
<permutate-expression> := 'PERMUTATE(' expression: <Any> ')
```

Since

3.0

Return-type

The return-type of the argument

Description

The current filter can contain more than one key for a dimension, i.e. several years or products. Usually a function is executed with these

multiple keys but for complexity reasons it can be useful to execute an expression with single keys only.

To do this, you can wrap any expression with the PERMUTATE function. It will permute all keys of all dimensions and call the function multiple times with the permuted filter. The result is a list of all execution results.

By default, formulas in a configuration are always permute unless the “permute” property of the formula is set to “false”. Therefore, this function is only useful in reports.

Examples

With a filter Time:'Jan/2011' | Time:'Feb/2011' | Product:A | Product:B the function:

```
Amount() / Quantity()
```

would divide four values by other four values, because the fact functions would return four values each. The result would be an error, because the DIV function only allows single divisors.

The function:

```
PERMUTATE( Amount() / Quantity() )
```

would call the division four times, always with one value and one divisor. The result would be four correct values.

See also

TUPLE

POSITIONOF

Syntax

```
<positionof-expression> := 'POSITIONOF(' keys: <Key> ')
```

Since

2.0

Return-type

Integer

Description

The function POSITIONOF determines the position of a key below his parent and returns it. The first child has always the position 0. Furthermore, the root key of a dimension constantly has the position 0.

If the argument contains more than one key, the corresponding number of positions is returned.

If the argument is NULL, the function returns NULL.

Examples

```
POSITIONOF( Time:'01.01.2011' )
```

= 0

```
POSITIONOF( Time:'03.01.2011' )
```

= 2

See also

LEVELOF

POW

Syntax

```
<pow-expression> := 'POW(' values: <Number> ',' exponent:
<Number> ')
```

Since

2.2.6

Return-type

Number

Description

Returns the value of the first argument raised to the power of the second argument.

If any argument is NULL, the function returns also NULL.

Examples

```
POW( 2, 3 )
```

= 2^3= 8

See also

SQRT

PRED

Syntax

```
<pred-expression> := 'PRED('
   keys: <Key>
   [ ',' parent-level: <Integer>
   [ ',' distance: <Integer>
   [ ',' count: <Integer> ] ] ] ')'
```

Since

2.2

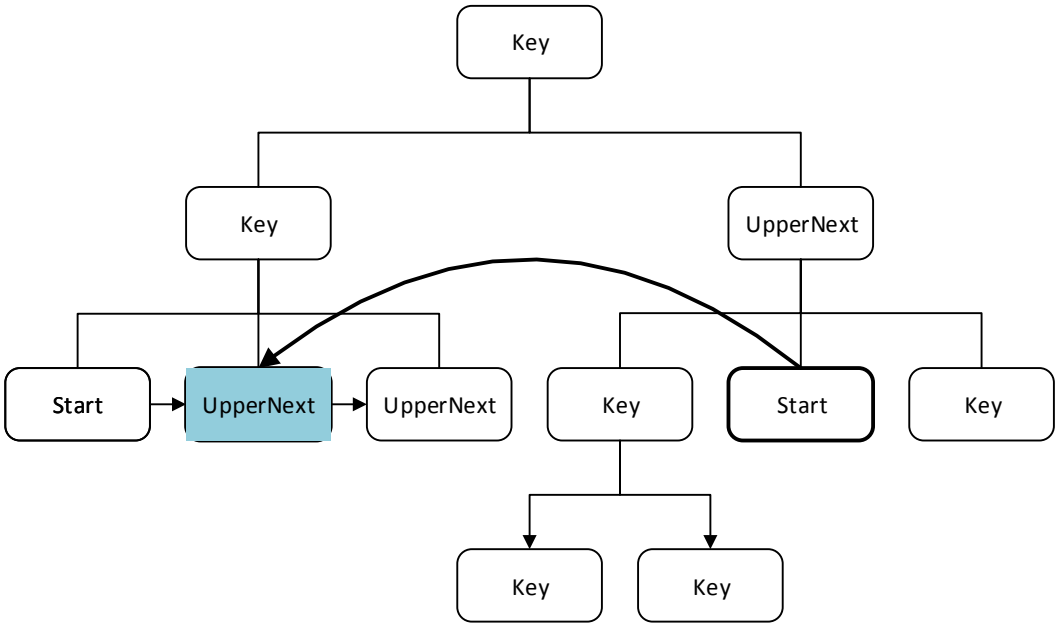
Return-type

Key

Description

Returns the predecessors of the keys passed as first argument. In difference to the function PREV, this function does not return the directly previous key. It searches for a similar child in the predecessor of the parent (or ancestor). This child will have the same position within its parent than the argument.

For instance, it can search for the same month in a previous year.



If the parent or other ancestor is used is defined by the second argument, which defines the level of the level. I.e. when searching for the same month in the previous year, the level should be 1 (YEAR). Then the month is searched in the predecessor of the YEAR.

Like in the PREV function, you can also define the distance (third argument) and the number of keys (fourth argument) you wish to return. This distance is applied to the parent key – for instance, in the upper previous example. A distance of 2 would search the same month for two years ago.

If any of the argument is NULL, this function returns NULL.

Examples

```
PRED( Time, 1 )
```

Returns the corresponding day, month etc. of the previous year

```
PRED( Time, 2 )
```

Returns the corresponding day of the previous month

```
PRED( Time, 1, 2 )
```

Returns the corresponding day, month etc. of the previous of the previous year

```
PRED( Time, 1, 1, 3 )
```

Returns the corresponding days, months etc. of the last 3 previous years

```
PRED( NULL, 1 )
```

= NULL

```
PRED( Time, NULL )
```

= NULL

See also

NEXT, PREV, SUCC, YTD

PREV

Syntax

```
<prev-expression> := 'PREV(' keys: <Key> [ ',' distance:
<Integer> [ ',' size: <Integer> ] ] ')'
```

Since

1.0

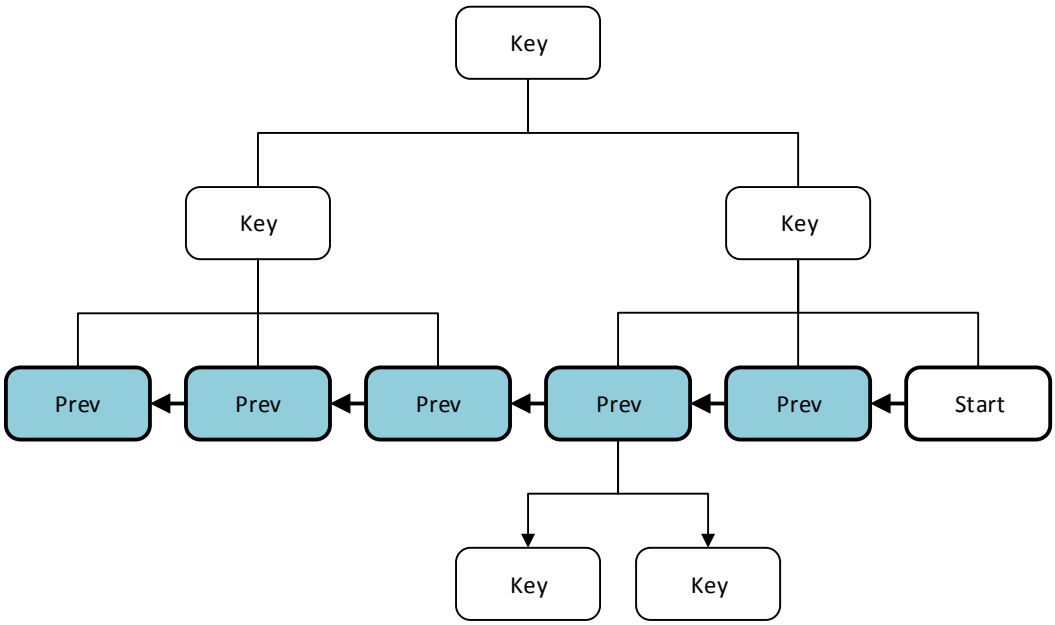
Return-type

Key

Description

PREV evaluates the preceding keys of the keys from the first argument. The predecessor of a key is the previous key within the same level of the dimension. It must not have compellingly the equivalent father like the key itself.

If no distance (with argument 2) is defined, the directly preceding key is returned. If a distance N is defined, the key with the distance N from the key is determined. I.e. “PREV(Time, 2)” corresponds to “PREV(PREV(Time))”.



With the third argument, you can also define the number of keys you want the function to return. The default value is 1.

If the argument is NULL, the result is NULL. If the keys have no predecessor, because they are first within the level, the result empty.

Examples

```
PREV( Time:'Dec/2011' )
```

= Time:'Nov/2003'

```
PREV( Time:'Dec/2011', 3 )
```

= Time:'Sep/2003'

```
PREV( Time:'Jan/2011' )
```

= NULL

```
PREV( NULL )
```

= NULL

See also

NEXT, PRED, SUCC, UPPERNEXT, UPPERPREV, YTD

RANGE

Syntax

```
<range-expression> := 'RANGE(' from: <Key> ',' to: <Key> ')'
```

Since

2.2.2

Return-type

Key

Description

This function returns the two keys from the arguments and all keys between them. If both keys belong to different dimensions, the function will return an empty result.

If both keys belong to the same dimension but have different levels, the result will not contain the key with the higher level but its first child (or descendant) within the same level.

[image]

If the position of the second argument within the dimension is lower than the position of the first, the function will swap them. The result always starts with the key with the first position.

The function also accepts more than one key for each argument. Then it will return multiple ranges, one for each combination of start- and end-key.

If any of the arguments is NULL, the function will return also NULL.

Examples

```
RANGE( Time:'Jan/2011', Time:'Apr/2011' )
```

= Time:'Jan/2011' | Time:'Feb/2011' | Time:'Mar/2011' | Time:'Apr/2011'

```
RANGE( Time:'Apr/2011', Time:'2011' )
```

= Time:'Jan/2011' | Time:'Feb/2011' | Time:'Mar/2011' | Time:'Apr/2011'

```
RANGE( Time:'Jan/2011', NULL )
```

= NULL

See also

CLUSTER, NUMBER_RANGE

REGEXP

Syntax

```
<regexp-expression> := 'REGEXP(' text: <String> ','
expression: <String> ')'
```

Since

2.5

Return-type

String

Description

The REGEXP function parses the first argument with the regular expression defined by the second arguments and returns all text fragments of the first arguments matching the expression.

If any argument is NULL, the function will return NULL.

Examples

```
COUNT( REGEXP( 'Hello world', 'l|o' ) )
```

= 5 (the number of occurrences of the characters 'l' and 'o' in the argument)

```
COUNT( REGEXP( 'Hello world', NULL ) )
```

= NULL

See also

LIKE

REGRESSION

Syntax

```
<regression-expression> := 'REGRESSION(' expression: <Number>
',' source-keys: <Key> ',' target-key: <Key> [ ',' <Integer> ]
')'
```

Since

2.2

Return-type

Double

Description

Calculates the regression, based on expression, for a single target key based on multiple input keys:

- The first argument defines the expression you want to calculate the regression with. This can be a simple fact, i.e. Amount(), or any other complex expression.
- The second argument is a number of keys, which are used as input for the calculation. This can be a whole dimension level or particular keys, i.e. the months of the last year.
- The third argument defines the target key the regression, e.g. the current month.

Like the FORECAST function, this function only calculates a single value for a single target key. If you want to create a curve with this function, you can use it as formula and evaluate the regression for each cell of your table or graph..

The optional 4th argument allows to shift the result up or down to the upper or lower border of the regression (then the regression-line will touch the most upper or lower point of the input-data set). To shift the regression up, pass a value > 0 here. To shift it down, pass any value < 0.

Examples

```
REGRESSION( Amount(), LEVEL( Time, 3 ), Time )
```

Uses all existing values of Amount for the third-time level to calculate the regression for the current Time key.

See also

FORECAST, SPLINE, VARIANCE

REPLACE

Syntax

```
<replace-expression> := 'REPLACE(' text: <String> ',' old-
pattern: <String> ',' new-pattern: <String> ')'
```

Since

2.0

Return-type

String

Description

The REPLACE-function replaces all occurrences of the string defined as argument two in argument one by the string defined in argument three.

If any argument is NULL, the function returns NULL.

Examples

```
REPLACE( 'Hello World', 'World', 'Customer' )
```

= 'Hello Customer'

See also

CONTAINSTEXT, LIKE, SUBSTR

RETURNTYPE

Syntax

```
<returntype-expression> := 'RETURNTYPE(' expression: <Any> ')'
```

Since

2.2.0

Return-type

String

Description

This debugging-function returns the name of the return-type passed as the argument. The function does not execute the expression, only its return-type is determined.

Examples

```
RETURNTYPE( 10 )
```

= 'Integer'

```
RETURNTYPE( NULL )
```

= 'All'

```
RETURNTYPE( NEXT( Time ) )
```

= 'Time'

See also

DEBUG, TYPE

REVERSE

Syntax

```
<reverse-expression> := 'REVERSE(' values: <Any> ')'
```

Since

1.2

Return-type

The type of the argument.

Description

This function returns all values from the argument in reverse order.

Examples

```
REVERSE( 1 | 2 | 3 )
```

= 3 | 2 |1

```
REVERSE( LEVEL( Time, 3 ) )
```

I.e. returns Time:'Dec/2011' + ... + Time:'Jan/2011'

See also

DISTINCT, INTERSECT, JOIN, WITHOUT

RIGHT

Syntax

```
<right-expression> := 'RIGHT(' text: <String> ',' size:
<Integer> ')'
```

Since

2.0

Return-type

String

Description

The function RIGHT returns the N (defined by the second argument) right characters of the string passed as first argument. If the length of a string is smaller than N, the original string is returned.

If any argument is NULL, the function returns NULL.

Examples

```
RIGHT( 'Hello world', 5 )
```

= 'world'

```
RIGHT( NULL, 5 )
```

a= NULL

```
RIGHT( 'Hello world', NULL )
```

= NULL

See also

LEFT, SUBSTR, SUBSTRINGBEHIND

ROUND

Syntax

```
<round-expression> := 'ROUND(' value: <Number> ')'
```

Since

2.0

Return-type

Integer

Description

The function ROUND returns the closest integer to the argument. The function rounds to an integer by adding 1/2, taking the floor of the result, and casting the result to type integer.

If the argument is NULL, the function returns NULL.

Examples

```
ROUND( 0.5 )
```

= 1

```
ROUND( 0.4 )
```

= 0

```
ROUND( NULL )
```

= NULL

See also

CEIL, FLOOR

ROWNUM

Syntax

```
<rownum-expression> := 'ROWNUM( [ max-rows: <Integer> [ ',' first-row: <Integer> ] ] )'
```

Since

2.2

Return-type

Key (members of the LINE_DIMENSION)

Description

This function is needed to create detail reports, which show drill through data directly taken from data sources (i.e. SQL databases).

Because for drill through queries, there is nothing you can iterate on the Y-axis of a pivot-table to create rows (or on the X-axis to create columns), there is a system dimension named "LINE_DIMENSION". It contains an endless number of keys representing simple line-numbers. This function helps you to calculate how many line-numbers the query will need to display the complete result of the query.

It returns members of the LINE_DIMENSION (beginning with the first key LINE_DIMENSION:'1') which can then be used in the iteration of a pivot-table.

With both optional arguments, you can define the maximum number of rows you want to create (defined by the first argument) and the position of the first line you want to create (defined by the second argument). With these arguments, you can limit the result or create paging tables.

If any of these arguments is NULL, the functions return an empty set.

Examples

```
ROWNUM()
```

Returns the needed keys of the LINE_DIMENSION

```
ROWNUM( 100 )
```

Returns the first 100 keys of the LINE_DIMENSION (or less if there are less rows in the data-source).

```
ROWNUM( 100, 50 )
```

Returns 100 keys of the LINE_DIMENSION (or less) beginning with key 50.

ROWSPAN

Syntax

```
<rowspan-expression> := 'ROWSPAN(' x: <Integer> ')'
```

Since

2.2.2

Return-type

Integer

Description

This function returns the number of rows spanned by the header in the same row with the given X position (passed as argument). I.e. “ROWSPAN(0)” returns the span of the leftmost header.

For instance, you can use this function to create subtotals for tables with grouped headers.

Examples

```
SUM( TONUMBER( MATRIX( X(), Y() - 1, X(), Y() - ROWSPAN(0) ) )
)
```

Returns a subtotal for all rows spanned by the same, grouping header in the Y-axis.

See also

COLSPAN

RTRIM

Syntax

```
<rtrim-expression> := 'LTRIM(' text: <String> ')'
```

Since

2.2.6

Return-type

String

Description

The function TRIM removes all trailing white spaces from the input string passed as argument. If the argument is NULL the function also returns NULL.

Examples

```
RTRIM( ' Hello World ' )
```

= ' Hello World'

```
RTRIM( NULL )
```

= 'NULL

See also

LTRIM, TRIM

SCREENY

Syntax

```
<screeny-expression> := 'SCREENY()'
```

Since

3.1

Return-type

Integer

Description

This function returns the current Y position (row number) of the header or cell when this function is executed inside pivot tables. In difference to the Y function, this function returns the real Y position after empty rows were suppressed from the result (if Suppress Rows is enabled for the query).

If the function is executed outside a table, an error is raised.

Examples

```
background="IIF( SCREENY() % 2 = 0, 'gray', 'white' )"
```

For instance, you can use this expression to colorize header backgrounds (every second row becomes gray).

See also

MATRIX, MAX_X, MAX_Y, MIN_X, MIN_Y, X, XHEADER, Y, YHEADER

SORT

Syntax

```
<sort-expression> := 'SORT(' keys: <Key> ',' expression:
<Value> [ ',' size: <Integer> [ ',' < Boolean> ] ] ')'
```

Since

2.0

Return-type

Key

Description

The SORT function sorts a set of keys by any criteria. The first argument defines the keys to be sorted. The second argument is a value-expression which defines the sort criteria for the keys.

Like the FOREACH and MATCH functions, this function executes the criteria expression for each key of the keys with respect to the current filter. The keys then will be sorted by their criteria results.

With the (optional) third argument you can define a size-limit for your result. This enables you to pick the greatest or smallest N keys out of the key-set and to perform top 10 queries. The default-value for this argument is NULL, which doesn't limit the result-size.

The (optional) fourth argument defines the order of the result: If the argument is TRUE, the keys will be sorted by a descending order. If the argument is FALSE (default), the keys will be sorted in an ascending order.

Examples

```
SORT( LEVEL( Article, 3 ), Article.Color )
```

Returns all articles sorted by their colors

```
SORT( LEVEL( Article, 3 ), Amount(), 10, true )
```

Returns the top-10 articles sorted by their Amount

See also

FOREACH, LOOKUP, MATCH

SPLINE

Syntax

```
<spline-expression> := 'SPLINE(' expression: <Number> ','
target-key: <Key> ',' input-keys: <Key> ')'
```

Since

2.2

Return-type

Double

Description

This function calculates a spline graph for the expression defined by the first argument. It returns the value for the data-point defined by the second argument.

For the calculation of the graph, you'll also have to define an input-data set. This is done with the third argument, a number of keys for which the expression will be executed. The results of these calculations will then be taken as the input-data set.

Examples

```
SPLINE( Amount(), Time, LEVEL( Time, 3 ) )
```

Calculates the splined amount for the current time, based on the complete level 3 of the time dimension.

See also

FORECAST, REGRESSION

SPLIT

Syntax

```
<split-expression> := 'SPLIT(' text: <String> ',' delimiter:
<String> ')'
```

Since

2.0

Return-type

String

Description

The SPLIT function splits a string into parts. The first argument is the string to split. The second argument defines the delimiter by that the string should be split. The result is a number of strings without the delimiter.

If any argument is NULL, the function returns NULL.

Examples

```
SPLIT( 'Split,this,string', ',' )
```

= 'Split' | 'this' | 'string'

```
SPLIT( NULL, ',' )
```

= NULL

```
SPLIT( 'Split,this,string', NULL )
```

= NULL

See also

CONCAT, REGEXP

SQRT

Syntax

```
<sqrt-expression> := 'SQRT(' value: <Number> ')'
```

Since

2.6.0

Return-type

Number

Description

Returns the square root of the numbers passed as argument.

If NULL is passed as argument, the function returns NULL.

Examples

```
SQRT( 9 )
```

= 3

```
SQRT( NULL )
```

= NULL

See also

POW

STARTSWITH

Syntax

```
<startswith-expression> := 'STARTSWITH(' text: <String>,
pattern: <String> ')'
```

Since

2.1

Return-type

Boolean

Description

This function tests if the first argument starts with the string passed as second argument.

If any argument is NULL, the function returns NULL.

Examples

```
STARTSWITH( 'Hello World', 'World' )
```

= FALSE

```
STARTSWITH( 'Hello World', 'Hello' )
```

= TRUE

```
STARTSWITH( 'Hello World', 'NULL' )
```

= NULL

See also

CONTAINSTEXT, ENDSWITH

STRLEN

Syntax

```
<strlen-expression> := 'STRLEN(' text: <String> ')'
```

Since

2.0

Return-type

Integer

Description

This function returns the length of the string passed as arguments.

If the argument is NULL, the function returns NULL.

Examples

```
STRLEN( 'Hello world' )
```

= 11

```
STRLEN( NULL )
```

= NULL

See also

TEXTPOSITION

SUB

Syntax

```
<number-expression> := 'SUB(' value1: <Number> ',' value2:
<Number> ')'

<number-expression> := <Number> '-' <Number>
```

Since

1.0

Return-type

Number

Description

This function calculates the difference between argument 1 and argument 2.

If one of the arguments is NULL, the function returns NULL.

Instead of this function, you also can use the operator "-".

Examples

```
SUB( 10, 5 )
```

= 5

```
10 - 5
```

= 5

```
-10 - 20
```

= -30

```
SUB( 10, NULL )
```

= NULL

See also

ADD, DIV, MUL, SUM

SUBSTR

Syntax

```
<substr-expression> := 'SUBSTR(' text: <String> ',' begin-
index: <Integer> ',' end-index: <Integer> ')'
```

Since

2.0

Return-type

String

Description

The function SUBSTR returns a sub string of the string passed as first argument.

The sub string begins with the character at the index specified by the second argument and extends it to the character at the index defined by the third argument – 1. Thus, the length of the sub string is end index - begin index.

If any argument is NULL, the function returns NULL.

Examples

```
SUBSTR( 'Hello World', 0, 5 )
```

= 'Hello'

```
SUBSTR( NULL, 1, 5 )
```

= NULL

See also

LEFT, RIGHT, SUBSTRINGBEHIND

SUBSTRINGBEHIND

Syntax

```
<substringbehind-expression> := 'SUBSTRINGBEHIND(' text:
<String> ',' pattern: < String> ')'
```

Since

2.2.3

Return-type

String

Description

This function returns the part of the argument one behind the first occurrence of the second argument. If it does not occur, the function returns NULL.

If any argument is NULL, the function will return NULL.

Examples

```
SUBSTR( 'Hello World', ' ' )
```

= 'World'

See also

LEFT, RIGHT, SUBSTR

SUBTOTAL

Syntax

```
<number-expression> := 'SUBTOTAL(' [ name: <String> [ ',' incremental: <boolean> ] ] ')'
```

Since

2.5

Return-type

Double

Description

This function can be used to add totals and subtotals to a pivot table. The function will summarize all previous values in the same column (if the function is used in a Y-header formula) or in the same row (if the function is used in a X-header formula) and returns the summarized value.

The first and optional argument allows to define a name for the subtotal. With this name, you can use the function multiple times in the same row or column, because it will not summarize values generated by the SUBTOTAL function with different name. If no name is defined, the system uses the name 'DEFAULT'.

The second and also optional argument determines, if the subtotal is absolute (FALSE) or incremental (TRUE). 'Absolute' means, it only adds the values since the last row or column the SUBTOTAL was calculated (if the variable name is equal). The 'incremental' mode will add the last SUBTOTAL value plus all new values since them.

The default value for this argument is 'FALSE' (absolute).

Examples

```
SUBTOTAL()
```

A simple subtotal

```
SUBTOTAL('CATEGORY')
```

A simple subtotal stored with the variable CATEGORY

```
SUBTOTAL('CATEGORY', true)
```

Incremental subtotal

See also

SUBTOTALS

SUBTOTALS

Syntax

```
<subtotals-expression> := 'SUBTOTALS(' keys: <Key> ')'
```

Since

2.2

Return-type

Key

Description

This function inserts additional keys to the argument in order to add summary lines to an iteration:

After each key, the parent key is added if the following key is not a child of the same parent. Furthermore, the parent of the parent is added if it changes and so on.

As a result, all parent keys are added to the result, with the top most all keys as last.

This function is very similar to the PEDIGREE function, but the higher aggregated keys are inserted after the detailed keys.

If the argument is NULL, the function returns NULL.

Examples

```
SUBTOTALS( LEVEL( TIME, 3 ) )
```

Returns all days with additional keys for months, years and the total.

See also

ANCESTORS, PEDIGREE, SUBTOTAL

SUCC

Syntax

```
<succ-expression> := 'SUCC('
  keys: <Key>
  [ ',' parent-level: <Integer>
  [ ',' distance: <Integer>
  [ ',' count: <Integer> ] ] ] ')'
```

Since

2.2

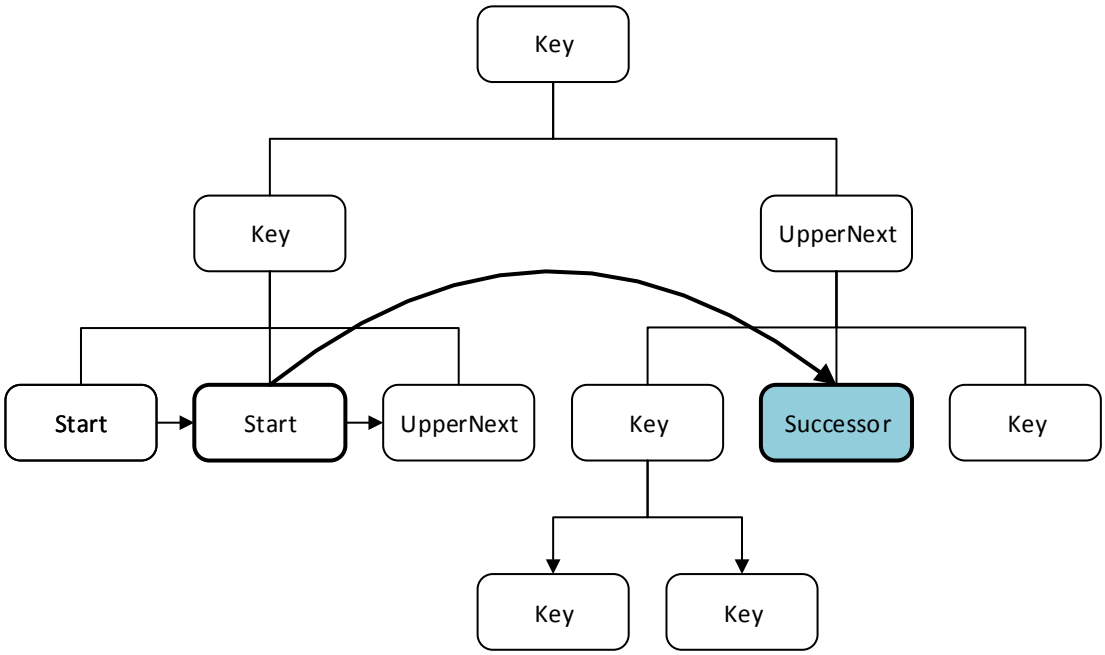
Return-type

Key

Description

Returns the successors of the keys passed as first argument. In difference to the function NEXT, this function does not return the directly next key. It searches for a similar child in the successors of the parent (or ancestor). This child will have the same position within its parent than the argument.

For instance, it can search for the same month in a following year.



If the parent or other ancestor is used is defined by the second argument, which defines the level of the level. I.e. when searching for the same month in the next year, the level should be 1 (YEAR). Then the month is searched in the predecessor of the YEAR.

Like in the NEXT function, you can also define the distance (third argument) and the number of keys (fourth argument) you wish to return. This distance is applied to the parent key – for instance, in the upper previous example, a distance of 2 would search the same month for two years ahead.

If any argument is NULL, this function returns NULL.

Examples

```
SUCC( Time, 1 )
```

Returns the corresponding day, month etc. of the next year

```
SUCC( Time, 2 )
```

Returns the corresponding day of the next month

```
SUCC( Time, 1, 2 )
```

Returns the corresponding day, month etc. of the next after the next year

```
SUCC( Time, 1, 1, 3 )
```

Returns the corresponding days, months etc. of the next 3 following years

```
SUCC( NULL, 1 )
```

= NULL

```
SUCC( Time, NULL )
```

= NULL

See also

NEXT, PRED, PREV

SUM

Syntax

```
<sum-expression> := 'SUM(' values: <Number> { ',' values:
<Number> } ')'
```

Since

2.0.3

Return-type

Double

Description

This function returns the sum of all values passed as arguments.

In difference to the ADD-function, the SUM-function only accepts and returns numerical values.

If any argument is NULL, this function returns the sum of all other values. If all arguments are NULL, it returns NULL.

Examples

```
SUM( 10, 20 )
```

= 30

```
SUM( Amount( CHILDREN( Product ) ) )
```

Returns the sum of all amounts of the sub-products

See also

ADD, AVG, DIV, MAX, MIN

SWITCH

Syntax

```
<switch-expression> := 'SWITCH(' test-value: <Any> { ',' 
match-value: <Any> ',' return-value: <Any> } ')'
```

Since

2.5.0

Return-type

The common super type of all return-value arguments.

Description

The SWITCH function calculates the first argument and then compares it to all match values of the following match / return value pairs.

With the first equal match value, the function will return the corresponding return value and stop the evaluation. If no match value matches, the function returns NULL.

The SWITCH function is similar to the CASE function but only calculates the first value and then compares w other (mostly constant) arguments. Therefore, it is usually faster and easier to use.

If the first argument is NULL, the function will return NULL without comparing any values.

Examples

```
SWITCH(
  Project.State,
  0, 'Not begun',
  1, 'In progress',
  2, 'Finished' )
```

Translates the numerical project-state to texts.

See also

CASE, IIF

SYSDATE

Syntax

```
<sysdate> := 'SYSDATE()'
```

Since

2.5

Return-type

Date

Description

The function SYSDATE returns the current system date.

See also

NOW, TIMESTAMP, TODATE

TEXTPOSITION

Syntax

```
<textposition-expression> := 'TEXTPOSITION(' text: <String>,
pattern: <String> ')'
```

Since

2.2

Return-type

Integer

Description

If the text second text is contained in the first argument, this function returns the position of its first character, beginning with 0 for the first possible position.

If the text is not contained, the function returns NULL. Furthermore, if any a is NULL, the function will return NULL.

Examples

```
TEXTPOSITION( 'Hello world', 'Hello' )
```

= 0

```
TEXTPOSITION( 'Hello world', 'world' )
```

= 6

```
TEXTPOSITION( 'Hello world', 'planet' )
```

= NULL

```
TEXTPOSITION( NULL, 'world' )
```

= NULL

```
TEXTPOSITION( 'Hello world', NULL )
```

= NULL

See also

CONTAINSTEXT, SUBSTR, SUBSTRINGBEHIND

TIMESTAMP

Syntax

```
<timestamp-expression> := 'TIMESTAMP(' format: <String> ')'
```

Since

1.2

Return-type

String

Description

This function returns the current date and time as string. The format is defined by the date-format-pattern passed as argument.

If the format is NULL, the function returns NULL.

Examples

```
TIMESTAMP( 'yyyy' )
```

= '2011'

```
TIMESTAMP( 'yyyyMMddHHmm' )
```

= '201108011321'

See also

NOW, SYSDATE, TODATE

TODATE

Syntax

```
<todate-expression> := 'TODATE(' value: <Any> [ ',' pattern:
<String> ] ')'
```

Since

2.2

Return-type

Date

Description

This function converts the value passed as argument (if possible) to a date:

- If the argument is a string, it is parsed. For parsing, the date-format defined by the second argument is used. If no format is defined, the function uses the standard date format depending on the current locale settings.
- If the argument is a date, the function simply returns the value.
- If the argument is a number, the value interpreted as milliseconds and used to construct a date.
- For any other type, the function returns NULL.

If the argument is NULL, the function returns NULL.

Examples

```
TODATE( '01.08.2011', 'dd.MM.yyyy' )
```

= 01.08.2011

```
TODATE( 0 )
```

= 01.01.1970

```
TODATE( NULL )
```

= NULL

See also

NOW, SYSDATE, TIMESTAMP, TOINTEGER, TONUMBER, TOSTRING

TOINTEGER

Syntax

```
<tointeger-expression> := 'TOINTEGER(' value: <Any> ')'
```

Since

2.2

Return-type

Integer

Description

This function converts the value passed as argument (if possible) to an integer value.

- If the argument is an integer, it is returned unchanged.
- If the argument belongs to any other number-type, it is returned without its fraction.
- If the argument is a string, it is parsed. If the strings contain a fraction, this fraction will be ignored.
- For any other type, the function returns NULL.

If the argument is NULL, the function returns NULL.

Examples

```
TOINTEGER( 42 )
```

= 42

```
TOINTEGER( 42.2 )
```

= 42

```
TOINTEGER( '42.2' )
```

= 42

```
TOINTEGER( NULL )
```

= NULL

See also

TODATE, TONUMBER, TOSTRING

TOKEY

Syntax

```
<tokey-expression> := 'TOKEY(' value: <Any> ')'
```

Since

2.2.2

Return-type

Key

Description

This function filters and returns the keys and NULL values of the argument and ignores all other values.

Examples

```
TOKEY( Time:'2011' )
```

= Time:'2011'

```
TOKEY( Time:'2011' | 100 )
```

= Time:'2011'

```
TOKEY( EVAL( 'Time::$LEVEL' ) )
```

= The level $LEVEL of the Time dimension

```
TOKEY( NULL )
```

= NULL

See also

TODATE, TONUMBER, TOSTRING

TOLOWER

Syntax

```
<tolower-expression> := 'TOLOWER(' text: <String> ')'
```

Since

1.2

Return-type

String

Description

Converts all the characters in the text to lower case. If the argument is NULL, the function returns NULL.

Examples

```
TOLOWER( 'Hello World' )
```

= 'hello world'

See also

BEAUTIFY, TOUPPER

TONUMBER

Syntax

```
<tonumber-expression> := 'TONUMBER(' value: <Any> ')'
```

Since

1.2

Return-type

Number

Description

This function converts the value passed as argument (if possible) to a number:

- If the argument is a string, it is parsed. Depending on the string, the returned value will be an integer or double value or NULL (if the string is not convertible)
- If the argument is a number, it is returned.
- For any other type, the function returns NULL.

- If the argument is NULL, the function returns NULL.

Examples

```
TONUMBER( '10' )
```

= 10

```
TONUMBER( '10.0' )
```

= 10

```
TONUMBER( '10.2' )
```

= 10.2

```
TONUMBER( 10 )
```

= 10

```
TONUMBER( 'Hello World' )
```

= NULL

```
TONUMBER( NULL )
```

= NULL

See also

TODATE, TOINTEGER, TOKEY, TOSTRING

TOSTRING

Syntax

```
<tostring-expression> := 'TOSTRING(' value: <Any> [ ',' 
format: <String> ] ')'

<tostring-expression> := '{' <Any> '}'
```

Since

1.1

Return-type

String

Description

The TOSTRING function converts arguments of any type into a string. According to the type of the argument, different conversions are used:

- Strings are returned without any modification.
- Integer-values are converted into a text without post comma places.
- Double-values are converted into a text with post comma places.
- Boolean-values result to 'TRUE' or 'FALSE'
- Date-values are converted to strings using the default locale format.
- Keys are converted to their ID.
- NULL results to NULL.

With the (optional) second argument you can refine the conversion: By passing a decimal or date format as second argument, you can determine exactly the format of converted numbers or dates. Depending on the value-type this must be a date- or number-format.

Instead of using the TOSTRING function you can also wrap any expression in the brackets “{“ and “}”.

Examples

```
TOSTRING( 10 )
```

= '10'

```
{10}
```

= '10'

```
TOSTRING( 10.2 )
```

= '10.2'

```
TOSTRING( 10.2, '0.00' )
```

= '10.20'

```
TOSTRING( Time:'Jan/2011' )
```

= 'Jan/2011'

```
TOSTRING( SYSDATE(), 'yyyy/MM' )
```

= '2011/08'

```
TOSTRING( TRUE )
```

= 'TRUE'

```
TOSTRING( NULL )
```

= NULL

See also

TODATE, TOINTEGER, TOKEY, TONUMBER

TOUPPER

Syntax

```
<toupper-expression> := 'TOUPPER(' text: <String> ')'
```

Since

1.2

Return-type

String

Description

Converts all characters in the text to be uppercase. If the argument is NULL, the function returns NULL.

Examples

```
TOUPPER( 'Hello World' )
```

= 'HELLO WORLD'

```
TOUPPER( NULL )
```

= NULL

See also

BEAUTIFY, TOLOWER

TRIM

Syntax

```
<trim-expression> := 'TRIM(' text: <String> ')'
```

Since

2.2.6

Return-type

String

Description

The function TRIM omits all leading and trailing white spaces from the text passed as argument. If the argument is NULL the function also returns NULL.

Examples

```
TRIM( ' Hello World ' )
```

= 'Hello World'

See also

LTRIM, RTRIM

TUPLE

Syntax

```
<toupper-expression> := 'TUPLE(' [ <Key> { ',' <Key> } ')'
```

Since

2.5

Return-type

Coordinate

Description

This function creates permutations from all keys passed as arguments and returns them as Coordinates.

Examples

```
TUPLE( Period:2008, Customer:'A' | Customer:'B' )
```

= (Period:2008,Customer:A) | (Period:2008,Customer:A)

See also

PERMUTATE

TYPE

Syntax

```
<type-expression> := 'TYPE(' expression: <Any> ')'
```

Since

2.2

Return-type

String

Description

This function is for debugging purpose. It returns the name of the type of the value returned by the argument (the name of the Java-class representing the type). In difference to the RETURNTYPE function, this function really executed the expression returns the real type of the value.

If the argument is NULL, the function returns NULL.

Examples

```
TYPE( 'Hello world' )
```

= 'java.lang.String'

```
TYPE( NULL )
```

= NULL

See also

DEBUG, RETURNTYPE

UNEQUAL

Syntax

```
<unequal-expression> := 'UNEQUAL(' value1: <Any> ',' value2:
<Any> ')'

<unequal-expression> := <Any> '<>' <Any>
```

Since

1.0

Return-type

Boolean

Description

This function checks whether the two arguments are unequal. This function returns the negated result of the EQUAL function. If any argument is NULL, this function returns NULL.

Instead of this function, you can also use the operator "<>".

Examples

```
UNEQUAL( 10, 20 )
```

= TRUE

```
10 <> 20
```

= TRUE

```
UNEQUAL( 10, 10 )
```

= FALSE

```
UNEQUAL( 10, 'Hello World' )
```

= TRUE

```
UNEQUAL( 10, NULL )
```

= NULL

See also

EQUAL, GREATER, GREATER_OR_EQUAL, LESS, LESS_OR_EQUAL, LIKE

UNIT

Syntax

```
<unit-expression> := 'UNIT(' fact: <Key> ')'
```

Since

2.2.1

Return-type

String

Description

Returns the unit name(s) of the fact(s) passed as argument.

If the argument is NULLor if the keys passed as arguments are no facts or if the facts don't have any unit, the function returns NULL.

Examples

```
UNIT( Fact:Turnaround )
```

= 'EUR'

```
UNIT( NULL )
```

= NULL

```
UNIT( Time:2006 )
```

= NULL

See also

FACTROOT

UPPERNEXT

Syntax

```
<uppernext-expression> := 'UPPERNEXT(' keys: <Key> ')'
```

Since

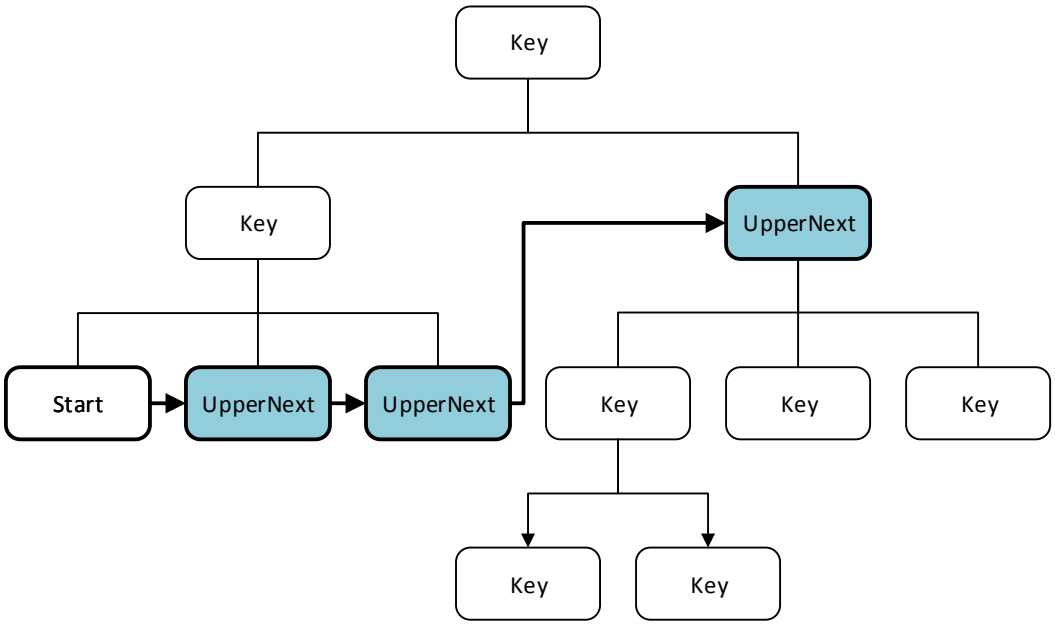
1.1

Return-type

Key

Description

The function UPPERNEXT determines, just as the function NEXT, the successor of a key. However, in contrast to the function NEXT, for the last child of a parent not the successor of the key but the successor of the parent is returned.



This function and the UPPERPREV function are very useful for generating fast year-to-date aggregates, because you can use higher aggregated levels for your calculations.

If the argument is NULL, this function returns NULL.

Examples

```
UPPERNEXT( Time:'Nov/2010' )
```

= Time:'Dec/2010'

```
UPPERNEXT( Time:'Dec/2010' )
```

= Time:'2011'

```
UPPERNEXT( NULL )
```

= NULL

See also

NEXT, PREV, UPPERPREV

UPPERPREV

Syntax

```
<upperprev-expression> := 'UPPERPREV(' keys: <Key> ')'
```

Since

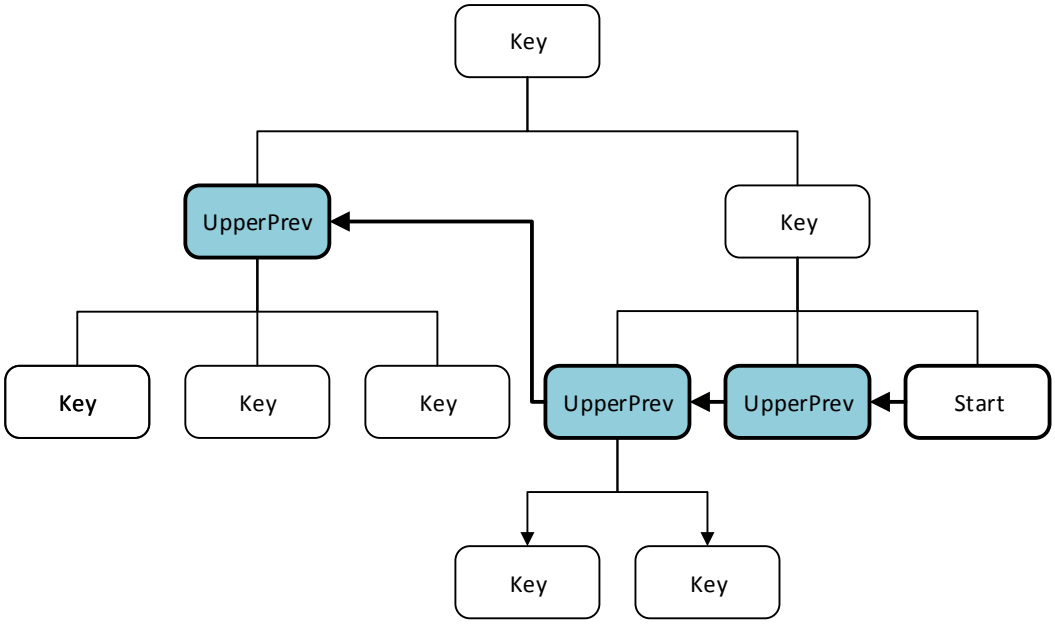
1.1

Return-type

Key

Description

The function UPPERPREV determines, just as the function PREV, the predecessor of a key at its level. However, in contrast to the function PREV, for the last child of a parent not the predecessor of the key but the predecessor of the parent is returned.



This function and the UPPERNEXT function are very useful for generating year-to-date aggregates, because you can use higher aggregated levels for your calculations.

If the argument is NULL, this function returns NULL.

Examples

```
UPPERPREV( Time:'Feb/2011' )
```

= Time:'Jan/2011'

```
UPPERPREV( Time:'Jan/2011' )
```

= Time:'2010'

```
UPPERPREV( NULL )
```

= NULL

See also

NEXT, PREV, UPPERNEXT

USER

Syntax

```
<User-Expression> := 'USER()'
```

Since

2.0

Return-type

String

Description

The function USER returns the name of the current user. Using this function is the same as using the variable $USER.

Examples

```
USER()
```

I.e. returns 'admin' (means the current user is 'admin')

See also

HASROLES, HASUSER

VARIANCE

Syntax

```
<variance-expression> := 'VARIANCE(' values: <Number> { ',' values: <Number> } ')'
```

Since

2.2

Return-type

Double

Description

It calculates the variance for all values of all arguments. If all values are NULL, the function returns NULL. Otherwise, all NULL values are ignored.

Examples

```
VARIANCE( 10, 15, 7, NULL, 10 )
```

= 11.0

```
NULL( NULL )
```

= NULL

See also

DEVIATION, PERCENTILE, REGRESSION

WITHOUT

Syntax

```
<without-expression> := 'WITHOUT(' value: <Any> ',' to-remove:
<Any> ')'
```

Since

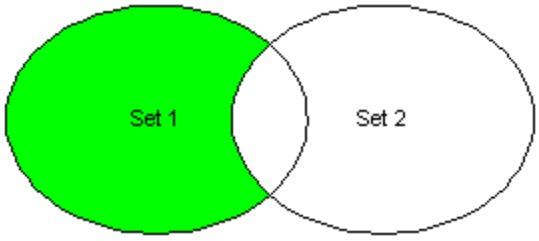
1.2

Return-type

Common super type of both arguments.

Description

The function WITHOUT returns all values from the first argument which are parts of the second argument.



If any argument is NULL, the function returns NULL.

Examples

```
WITHOUT( Time:MONTH ), Time:'Jan/2011' )
```

Returns all months except January 2011

See also

INTERSECT, JOIN

XML2ASCII

Syntax

```
<xml2ascii-expression> := 'XML2ASCII(' text: <String> ')
```

Since

2.6.0

Return-type

String

Description

This function removes all XML or HTML markup elements from the text and returns the rest as string.

If the argument is NULL, the function returns NULL.

Examples

```
XML2ASCII( "<p style='color: red;'>Hello <b>world</b></p>")
```

= "Hello world"

See also

BEAUTIFY

X

Syntax

```
<x-expression> := 'X()'
```

Since

2.0

Return-type

Integer

Description

This function returns the current X position (column number) of the header or cell when this function is executed inside pivot tables. If the function is executed outside a table, an error is raised.

Examples

```
background="IIF( X() % 2 = 0, 'gray', 'white' )"
```

For instance you can use this expression to colorize header backgrounds.

See also

MATRIX, MAX_X, MAX_Y, MIN_X, MIN_Y, SCREENY, XHEADER, Y, YHEADER

XHEADER

Syntax

```
<xheader-expression> := 'XHEADER(' pattern: <String> ')'
```

Since

2.0

Return-type

Integer

Description

With the XHEADER function, you can search the position of one or more X headers inside the current pivot table. The argument is a search pattern, which compared to the text of all headers on the x-axis. The result is a list of integers, representing the positions of all matching headers, or NULL if no header matches.

In the pattern, you can use the wild-cards '*' and '?' for specifying the headers name. If the pattern is NULL the functions returns NULL.

The usage of this function outside pivot-tables is invalid.

Examples

```
XHEADER( 'Sum' )
```

Returns the position of the header(s) with the Text 'Sum'

```
XHEADER( 'Article*' )
```

Returns the positions of all headers starting with 'Article'

See also

MATRIX, MAX_X, MAX_Y, MIN_X, MIN_Y, SCREENY, X, Y, YHEADER

Y

Syntax

```
<y-expression> := 'Y()'
```

Since

2.0

Return-type

Integer

Description

This function returns the current Y position (row number) of the header or cell when this function is executed inside pivot tables. If the function is executed outside a table, an error is raised.

Examples

```
background="IIF( X() % 2 = 0, 'gray', 'white' )"
```

For instance you can use this expression to colorize header backgrounds.

See also

MATRIX, MAX_X, MAX_Y, MIN_X, MIN_Y, SCREENY, X, XHEADER, YHEADER

YHEADER

Syntax

```
<yheader-expression> := 'YHEADER(' pattern: <String> ')'
```

Since

2.0

Return-type

Integer

Description

With the YHEADER function, you can search the position of one or more Y headers inside the current pivot table. The argument is a search pattern, which compared to the text of all headers on the y-axis. The result is a list of integers, representing the positions of all matching headers, or NULL if no header matches.

In the pattern, you can use the wild-cards '*' and '?' for specifying the headers name. If the pattern is NULL the functions returns NULL.

The usage of this function outside pivot-tables is invalid.

Examples

```
YHEADER( 'Sum' )
```

Returns the position(s) of the header with the Text 'Sum'

```
YHEADER( 'Article*' )
```

Returns the positions of all headers starting with 'Article'

See alsoMATRIX, MAX_X, MAX_Y, MIN_X, MIN_Y, SCREENY, X, XHEADER, Y

YTD

Syntax

```
<ytd-expression> := 'YTD(' keys: <Key> ',' ancestor: <Key> ')'
```

Since

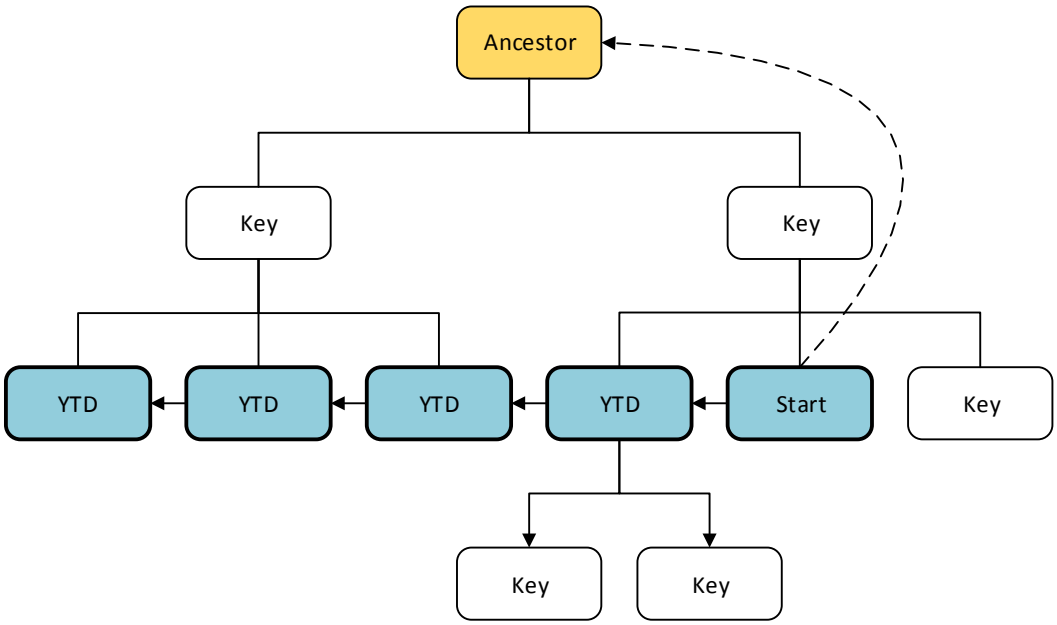
2.2

Return-type

Key

Description

The YTD function collects all predecessors of a key which belong to the same ancestor. The ancestor is defined with the second argument. The key itself is always added to the result.



If any argument is NULL, the function returns NULL.

You can use this function to build i.e. the summarized sales from the beginning of the current year, until now (therefore, YTD means year-to-date function).

Examples

```
SUM( Amount( YTD( Time, Time::YEAR ) ) )
```

Collects all Time keys with the same year (01.01 till now) and summarizes the amount.

See also

PRED, PREV, UPPERPREV, YTDR

YTDR

Syntax

```
<ytdr-expression> := 'YTDR(' keys: <Key> ',' ancestor: <Key>
')'
```

Since

3.0

Return-type

Key

Description

This function works very similar to the YTD function but has a different behavior when more than one key is passed as first argument.

The second argument will be recalculated for every single element of the first argument. In contrast, the YTD function calculates the second element once only and uses it a year to date parent for all keys.

When using the YTD in a function without the “Fast ToDo List” property set to “false” we recommend using this function.

See also

PRED, PREV, UPPERPREV, YTD

ZERO

Syntax

```
<zero-expression> := 'ZERO(' values: <Number> ')'
```

Since

1.1

Return-type

Number

Description

The function ZERO converts all NULL values from the argument into the integer-number "0". All other values are return without any manipulation.

Examples

```
ZERO( Amount() )
```

Returns 0 if the fact returns NULL, otherwise the fact value

```
AVG( Zero( Amount( LEAFS( Time ) ) ) )
```

Builds a real average over all days, counting days without values as 0.

See also

COALESCE, CUBE, EXISTS, ISNULL

I want to...

Reduce my tables to rows or columns containing all values

There are two ways to remove rows or columns with no data from a report: You can enable the property “Suppress Rows” or “Suppress Columns” in your report, or you can use different functions in the iteration of the table headers:

- The LOOKUP function helps you to find keys with data for a certain fact or expression. This is the standard function for suppressing rows or columns, because the properties still use more memory than the LOOKUP function.
- The SORT function allows to reduce the headers to the topmost X headers and to perform a top 10 analysis.
- The MATCH also allows to filter header iteration by any other criteria, for instance, a product manufacturer.

Sort headers or keys by a fact, attribute or expression

Like suppressing empty rows from a table, sorting a table is possible with functions or header properties:

- Either use the “Sort” property of a header to sort the rows or columns of your report. This will only sort the table but not remove any header.
- Alternatively, use the SORT function. Unless you don't want to reduce your table for a top 10 analysis, the SORT function is slower because it performs additional cube or database queries.

Perform a top 10 analysis

Use the SORT function to reduce the headers of your report to the topmost X elements for a certain fact or expression.

Filter keys

The MATCH function is a very power function. It allows to filter keys by any criteria or expression. Use the MATCH expression in the “Iteration” property of a header to remove all unwanted keys from your table axis.

Remove a “MULTIPLE VALUE” message from a report

Cells in a table can only display single values or stay empty. If you see a “MULTIPLE VALUE” message in a cell, this is caused by a filter with more than one key for one or more dimensions. Usually this is caused by “Multiple” selectors or “Group By” in a header.

To remove this message, you must aggregate all cell values to a single value:

- For numbers, use one of the aggregation functions AVG, MIN, MAX or SUM. Use them in the formula property of a header, for instance, use “SUM([])” to aggregate all values of the current cell.
- For String, you can concatenate multiple values with the CONCAT function.

Display the difference of two neighbor cells in %

With the function MATRIX, you can access neighbor cells. For instance, the expression

```
TONUMBER( MATRIX instance, ) ) / TONUMBER( MATRIX( X() - 1 ) )
```

divides the value two cells left from the current cell by the value left of the cell (use this expression in the “Formula” property of a header in the X axis and set the header format to “0%”).

The DEVIATION function is also a comfort function for calculating the difference of two values in percent.

Find the same month or day in a previous or next year

Sometimes you want to compare a value of the current month with the equal month of the last year.

The PRED allows to search for a similar key in a dimension. I.e. the iteration

```
PRED( Time, 1 )
```

displays the same month or day from the previous year (if the year has the level 1 in your time dimension).

Find the current year, month or date in a time dimension

When working with a time dimension, you may need to access the current day, month or year key of your time dimension.

The NOW function searches in a time dimension for a key which ID matches the current month, day or year.

Perform an ABC analysis

Use the ABC function to find all keys which values together build to top % of certain fact. For instance, you can find all products which turnaround in conjunction builds 80% of your companies turnaround with the following expression:

```
ABC( PRODUCT, 1.0, 0.2 )
```

Use a different background color for every 2nd row

The “Background” and “Cell Background” properties of headers allow to define the background color of the header itself the cells in the headers' row. Use a function which refers to the current row number to define a toggling color:

```
Background = "IIF(SCREENY() % 2 == 0, 'grey', NULL)"
```

Then do the same for the “Cell Background”. This function uses the modulo operator to figure out if the current row has an even number.

Note that it’s better to use SCREENY rather than Y, because the Y only returns the rows’ number before empty rows are removed from the result (if “Suppress Rows” is enabled).