SDMX Technical Working Group

VTL Task Force

# VTL – version 1.1
# (Validation & Transformation Language)

# Part 2 – Reference Manual

*(DRAFT FOR PUBLIC REVIEW)*

*October 2016*

# Foreword

The Task force for the Validation and Transformation Language (VTL), created in 2012-2013 under the initiative of the SDMX Secretariat, is pleased to present the draft version of VTL 1.1.

The SDMX Secretariat launched the VTL work at the end of 2012, moving on from the consideration that SDMX already had a package for transformations and expressions in its information model, while a specific implementation language was missing. To make this framework operational, a standard language for defining validation and transformation rules (operators, their syntax and semantics) had to be adopted, while appropriate SDMX formats for storing and exchanging rules, and web services to retrieve them, had to be designed. The present VTL 1.1 package is only concerned with the first element, i.e. a formal definition of each operator, together with a general description of VTL, its core assumptions and the information model it is based on.

The VTL task force was set up early in 2013, composed of members of SDMX, DDI and GSIM communities and the work started in summer 2013. The intention was to provide a language usable by statisticians to express logical validation rules and transformations on data, whether described as dimensional tables or as unit-record data. The assumption is that this logical formalization of validation and transformation rules could be converted into specific programming languages for execution (SAS, R, Java, SQL, etc.) but would provide a "neutral" expression at business level of the processing taking place, against which various implementations can be mapped. Experience with existing examples suggests that this goal would be attainable.

An important point that emerged is that several standards are interested in such a language. However, each standard operates on its model artefacts and produces artefacts within the same model (property of closure). To cope with this, VTL has been built upon a very basic information model (VTL IM), taking the common parts of GSIM, SDMX and DDI, mainly using artefacts from GSIM 1.1, somewhat simplified and with some additional detail. This way, existing standards (GSIM, SDMX, DDI, others) may adopt VTL by mapping their information model against the VTL IM. Therefore, although a work-product of SDMX, the VTL language in itself is independent of SDMX and will be usable with other standards as well. Thanks to the possibility of being mapped with the basic part of the IM of other standards, the VTL IM also makes it possible to collect and manage the basic definitions of data represented in different standards.

For the reason described above, The VTL specifications are designed at a logical level, independently of any other standard, including SDMX. The VTL specifications, therefore, are self-standing and can be implemented either on their own or by other standards (including SDMX). In particular, the work for the SDMX implementation of VTL is going in parallel to the work for designing the VTL 1.1 version, and will entail a future update of the SDMX documentation.

The first public consultation on VTL (version 1.0) was held in 2014. Many comments were incorporated in the VTL 1.0 version, published in March 2015. Other suggestions for improving the language, received afterwards, fed the discussion for building the present draft version 1.1, which contains many new features.

The VTL 1.1 package, containing the general VTL specifications independent of other standards possible implementations, will include, in its final release:

a) Part 1 – the user manual, highlighting the main characteristics of VTL, its core assumptions and the information model the language is based on;
b) Part 2 – the reference manual, containing the full library of operators ordered by category, including examples; this version will support more validation and compilation needs compared to VTL 1.0.
c) eBNF notation (extended Backus-Naur Form) which is the technical notation to be used as a test bed for all the examples.

The present document (part 2) contains the reference manual with the full library of operators ordered by category.

The latest version of VTL is freely available online at [https://sdmx.org/?page_id=5096](https://sdmx.org/?page_id=5096)

**Acknowledgements**

The VTL specifications have been prepared thanks to the collective input of experts from Bank of Italy, Bank for International Settlements (BIS), European Central Bank (ECB), Eurostat, ILO, INEGI-Mexico, ISTAT-Italy, OECD, Statistics Netherlands, and UNESCO. Other experts from the SDMX Technical Working Group, the SDMX Statistical Working Group and the DDI initiative were consulted and participated in reviewing the documentation.

The list of contributors and reviewers includes the following experts: Sami Airo, Foteini Andrikopoulou, David Barraclough, Luigi Bellomarini, Marc Bouffard, Maurizio Capaccioli, Vincenzo Del Vecchio, Fabio Di Giovanni, Jens Dossé, Heinrich Ehrmann, Bryan Fitzpatrick, Tjalling Gelsema, Luca Gramaglia, Arofan Gregory, Gyorgy Gyomai, Edgardo Greising, Dragan Ivanovic, Angelo Linardi, Juan Munoz, Chris Nelson, Stratos Nikoloutsos, Marco Pellegrino, Michele Romanelli, Juan Alberto Sanchez, Roberto Sannino, Angel Simon Delgado, Daniel Suranyi, Olav ten Bosch, Laura Vignola, Fernando Wagener and Nikolaos Zisimos.

Feedback and suggestions for improvement are encouraged and should be sent to the SDMX Technical Working Group ([twg@sdmx.org](mailto:twg@sdmx.org)).

# Table of contents

# Introduction

The VTL 1.1 library of the Operators is described hereinafter. The operators included in this version of VTL are summarized in the diagrams and tables below.
VTL 1.1 is made of two main parts: the VTL Definition Language (VTL-DL) and the VTL Manipulation Language (VTL-ML).

The former (VTL-DL) did not exist in VTL 1.0, because at that time VTL was intended to work on top of existing standards, like SDMX, DDI, GSIM or others, and therefore the definition of the artefacts to be manipulated (Data and their structures, Variables, Value Domains and so on) was assumed to be made using the implementing standards and not VTL itself. In other words, VTL 1.0 was not meant to define its artefacts and therefore only contained a manipulation language.

During the work for VTL 1.1, it was acknowledged as very recommendable and useful to have a complete definition language, able to define all the artefacts that VTL can manipulate. This is, first, to express structural and reusable definitions directly in VTL (even independently of other standards); second, to facilitate the use of VTL on top of other standards (through a proper mapping, the structural definitions of the other standards could be translated in VTL definitions and vice-versa); third, to make it possible to check at parsing time the coherency of the VTL manipulation expressions against the structure of the artefacts to be manipulated (even defined through VTL).

Therefore, VTL 1.1 has been equipped also with a definition language for VTL artefacts.

As for the manipulation language, VTL 1.0 contains a flat list of operators, in principle not related one another. A main suggestion for VTL 1.1 was to identify a core set of primitive operators able to express all the other operators of the language. This is in order to specify more formally the semantics of the available operators, avoiding possible ambiguities about their behaviour and fostering coherent implementations. The distinction between the core and standard library is mainly of interest of the VTL technical implementers.

The suggestion above has been acknowledged, so that the VTL 1.1 manipulation language is made of a core set of primitive operators and a standard library of derived operators, definable in term of the primitive ones. The standard library contains VTL 1.0 operators (possibly enhanced) and the new operators introduced in VTL 1.1.

The VTL core includes a mechanism called FLWOR expressions (For-Let-Where-Order-Return), which allows to define derived operators and their behaviour, including custom operators (not existing in the standard library) for specific purposes.

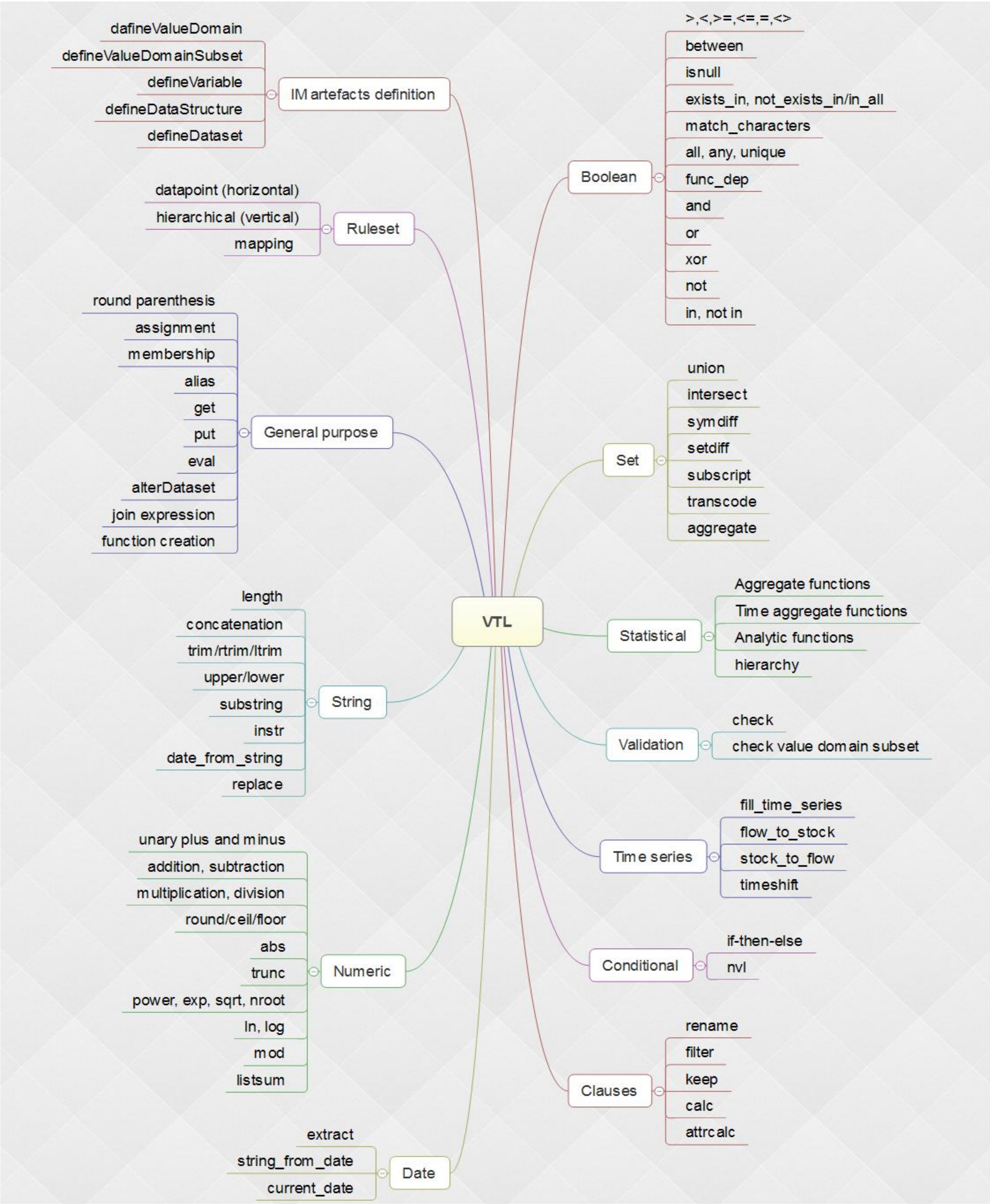
# Structure of the document

This manual describes in detail the operators of VTL 1.1 and is organized as follows.
In this chapter, the following paragraph (Diagrams of the Operators) summarizes all the available operators (for the VTL-DL, VTL-ML - Core Operators, VTL-ML – Standard Library) through a diagram. Then, in the paragraph "List of the Operators/Functions", two corresponding lists are given, specifying for each operator some basic information. In "Evaluation Order", the precedence rules for evaluation of the VTL-ML operators are described. Finally, the "Syntactical conventions" section illustrates the meta-syntax used in the other chapters for describing formally the syntax of the operators.
The remainder of the document is structured in chapters, each one dedicated to the description of a category of Operators. For each Operator there is a specific section explaining the syntax, the semantics and giving some examples. Each of these sections has the following structure:

- **Semantics**: an informal extract of the behaviour;
- **Syntax**: a specification of the complete syntax of the operator at hand. It is expressed in terms of the types of the Core (cfr. part 1) by means of a specific meta-syntax;
- **Parameters**: the input parameters, described in detail, with respect to the types of the Core;
- **Returns**: the output parameters, described in detail, with respect to the types of the Core;
- **Constraints**: semantic constraints and syntactical constraints that cannot be specified with the meta-syntax but need a textual explanation; sometimes for the sake of clarity, even syntactical constraints are also repeated.
- **Semantic specification**: an extensive explanation of the behaviour of the operator in terms of the syntactical elements described in the sections Syntax, Parameters and Returns. Sometimes, when particularly complex, specific constraints are explained also in this section.
- **Examples**: a series of examples proving the behaviour of the operator.

The last chapter illustrates the use case of a real questionnaire and the possible use of VTL for defining validation rules.

# Diagram of the Operators



# List of the Operators/Functions

The following tables list the VTL Operators and describe their main characteristics. The tables are relevant to the VTL-DL and the VTL-ML Standard Library. The operators of the Standard Library are ordered by category except for the clauses, which are the operators having a postfix syntax that are shown all together in the end.

The VTL-ML Standard Library includes operators that may act on both Data Set and on Structure Components of the Data Sets. The last column shows if the Operator acts on Dataset, Components or both, when meaningful. The Component version takes as input and returns in output Component expressions. They are part of the syntax of other operators or clauses, where specifically required for row-wise processing.

## VTL-DL Operators

| Operator/Functions | Category | Syntax | Description | Operand Data Sets | Component version |
|---|---|---|---|---|---|
| defineValueDomain | Information Model artefacts definition | Functional | defines a ValueDomain in VTL information model. | - | - |
| defineValueDomainSubset | Information Model artefacts definition | Functional | defines a ValueDomainSubset in VTL information model | - | - |
| Define Variable | Information Model artefacts definition | Functional | defines a persistent Variable in the VTL information model | - | - |
| defineDataStructure | Information Model artefacts definition | Functional | defines a persistent DataStructure in the information model | - | - |
| defineDataset | Information Model artefacts definition | Functional | defines a persistent Dataset in the information model | - | - |
| define datapoint ruleset | Ruleset | Functional | defines a persistent object that contains Rules to be applied to the Data Points | - | - |
| define hierarchical ruleset | Ruleset | Functional | defines a persistent object that contains Rules to be applied to the code items of a Dataset component. | - | - |
| define mapping | Ruleset | Functional | defines a persistent object that contains Rules to be applied to recode codes of a component in a Dataset | - | - |

# VTL-ML Standard Library

## Operators and functions applied on Datasets and scalar values

Most of the single data points operators and functions can be applied to both Datasets and scalar values. The operands of the operators and functions can take the following forms:

- Scalar expression, e.g. 1+2.
- Dataset expression, with a single measure or attribute selected using the membership operator ".", e.g. ds_bop.obs_value. In this case the operator or function is applied to the specified measure or attribute.
- Dataset expression, with no measure or attribute selected, e.g. ds_bop. In this case the operator or function is applied to all measures of the Dataset having the data type accepted by the operator.

When a VTL operator or function is applied to two or more Datasets then at least an Identifier Component must appear in all Datasets with the same name and data type. In this case the function is applied on the measures having the same name and data type (accepted by the operator) and for the matching data points, i.e., the data points that have the same values of the common Identifier Components.
Assuming that *f* is a VTL function or operator, *ds*, *ds1* and *ds2* are Datasets and *c* is a scalar value (constant), the following table shows the VTL rules in the case of binary operators or functions:

| Case | Result | Computation rule | Examples |
|---|---|---|---|
| *f* ( *c* , *c* ) | A scalar value | *f* is applied to the scalar operands | 1 + 1<br>round ( 10.52, 1)<br>"abc" \|\| "cde" |
| *f* ( *ds* , *c* ) | A Dataset having the same components Identifiers and Attributes Components) of *ds*. The Measure Components returned are only those having data type accepted by the operator. The other Measures will be discarded. | *f* is applied to all measures of *ds* having data type accepted by the operator.<br>*f* is applied to all data points of ds.<br>The cardinality of the resulting Dataset (number of data points) is the same of *ds*. | *ds* + 1<br>round ( *ds*, 1 ) |
| f ( *ds1* , *ds2* ) | A Dataset having all the Identifier Components (without duplicates) and the common measures of *ds1* and *ds2* having data type accepted by the operator. The other Measures will be discarded.<br>The attributes of *ds1* and *ds2* are ignored (do not appear in the resulting Dataset). | *f* is applied to the common numeric measures of ds1 and ds2.<br>*f* is applied to all matching data points of *ds1* and *ds2* (those having the same values of the common Identifier Components) and to the Measures having data type accepted by the operator.<br>The cardinality of the resulting Dataset (number of data points) is the number of matching data points. | *ds1* + *ds2*<br>mod ( ds1, ds2) |
| f ( *ds.m* , *c* ) | A Dataset having all the Identifier Components of *ds*, the specified Measure Component *m* and the Attribute Components of *ds*. | *f* is applied to the specified Measure Components of ds.<br>*f* is applied to all data points of ds. | round ( *ds*.obs_value, 1)<br>*ds*.obs_comment \|\| "." |

| | | | |
|---|---|---|---|
| | | The cardinality of the resulting Dataset (number of data points) is the same of ds. | |
| f ( *ds1.m1* , *ds2.m1* ) | A Dataset having all the Identifier Components (without duplicates) of *ds1* and *ds2*, and the Measure Component *m1*. The same Measure must be selected in both Datasets.<br>The Attributes of ds1 and ds2, and the other Measures (if any), are ignored (do not appear in the resulting Dataset). | f is applied to the specified Measure of ds, or to the common measures of ds.<br>f is applied to all matching data points of ds1 and ds2 (those having the same values of the common Identifier Components).<br>The cardinality of the resulting Dataset (number of data points) is the number of matching data points. | *ds1* + *ds2*<br>mod ( *ds1*.obs_value, *ds2*.obs_value) |

To apply the function *f* to measures having different names (in different Datasets) is possible using the operator **as**, e.g.:

ds1.obs_value + ( ds2.obsval as obs_value )

A Dataset contains a set of data points. A data point (statistical observation) can be thought of as a row in a relational table or as a cell in a hypercube.

Scalars are also supported. As we will show, many operators allow for a kind of hybrid combination, involving Datasets and scalars. In this case the scalar value is combined (according to the semantics of the operator at hand) with all the Data Points in the Dataset, and in particular with the respective values of the Measure Component.

For example:

ds2 := ds1 + 1

produces a Dataset ds2 with the same structure as ds1, where the constant numeric value 1 has been added to the value of the Measure Component of every single Data Point in ds1. Seen in another perspective, with this behavior, we propose a kind of implicit "promotion" of a scalar value into a somehow special Dataset, with one single Data Point, having one Measure Component (with the constant value) and with no Identifier Components.

In such a case, this single Data Point will match with all the Data Points of the involved Data Set as a limit but straightforward case, since, indeed, there are no Identifier Components to be matched at all.

## List of standard library operators and functions

| Operator/<br>Functions | Category | Syntax | Description | Operand Data Sets | Component version | Core/<br>Standard |
|---|---|---|---|---|---|---|
| Round parenthesis () | General purpose | Functional | Specifies the evaluation precedence | 1 | YES | Standard |
| assignment<br>:= | General purpose | Infix | Assigns an Expression to a model artefact | 2 | NO | Standard |
| Membership<br>. | General purpose | Infix | Identifies a Component within a Data Set | 1 | NO | Standard |
| Alias<br>as | General purpose | Infix | Define an alias for a component or for the result of an expression | 1 | YES (ONLY) | Standard |

| | | | | | | |
|---|---|---|---|---|---|---|
| alterDataset | General purpose | Functional | Modify the Dataset with all or a subset of input components having only the Identifier role | 1 | NO | Standard |
| get | General Purpose | Functional | Retrieves a Data Set | 1..N | NO | Standard |
| put | General Purpose | Functional | Stores a Data Set | 1 | NO | Standard |
| eval | General Purpose | Functional | Evaluates an external routine | 1 | NO | Standard |
| join expression | General Purpose | Functional | Implements the FLWOR expression | 1..N | YES | Core |
| Function creation | General Purpose | Functional | Creates a function | 1..N | YES | Core |
| null | General Purpose | Functional | null literal | 0 | YES | Core |
| length | String | Functional | Returns the length of a string | 1 | YES | Standard |
| concatenation \|\| | String | Functional | Concatenates two strings | 2 | YES | both |
| trim/ltrim/rtrim | String | Functional | Eliminates trailing or/and leading whitespace from a String | 1 | YES | Standard |
| upper/lower | String | Functional | Makes a string upper / lower case | 1 | YES | Standard |
| substr | String | Functional | Extracts a substring from a string | 1 | YES | Standard |
| instr | String | Functional | Returns the position of a String in another one | 1 | YES | Standard |

| | | | | | | |
|---|---|---|---|---|---|---|
| date_from_string | String | Functional | Change a string into a date | 1 | YES | Standard |
| replace | String | Functional | Replace a string with another one into a string | 1 | YES | Standard |
| unary plus<br>+ | Numeric | Infix | Leaves the sign unaltered | 1 | YES | both |
| unary minus<br>- | Numeric | Infix | Changes the sign | 1 | YES | both |
| addition<br>+<br>and subtraction<br>- | Numeric | Infix | Sum or subtract two numbers | 2 | YES | both |
| multiplication<br>*<br>and division<br>/ | Numeric | Infix | Multiply or divide two numbers | 2 | YES | both |
| round/ceil/floor | Numeric | Functional | Rounds a number | 1 | YES | Standard |
| abs | Numeric | Functional | Calculates the absolute value | 1 | YES | Standard |
| trunc | Numeric | Functional | Truncates the values | 1 | YES | Standard |
| exp | Numeric | Functional | Calculates the exponential | 1 | YES | both |
| ln | Numeric | Functional | Calculates the natural logarithm | 1 | YES | Standard |
| log | Numeric | Functional | Calculates the a base b logarithm | 1 | YES | Standard |
| power | Numeric | Functional | Calculates the power | 1 | YES | Standard |
| sqrt | Numeric | Functional | Calculates the square root of a number | 1 | YES | Standard |
| nroot | Numeric | Functional | Calculates the | 1 | YES | Standard |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | n-th root | | | |
| mod | Numeric | Functional | Calculates the modulo | 1 | YES | both |
| listsum | Numeric | Functional | Sums numbers replacing null with zero | 1..N | YES | Standard |
| equal to<br>= | Boolean | Infix | Compares the values | 2 | YES | both |
| not equal to<br><> | Boolean | Infix | Compares the values | 2 | YES | both |
| Greater than<br>>, >= | Boolean | Infix | Compares the values | 2 | YES | both |
| Less than<br><, <= | Boolean | Infix | Compares the values | 2 | YES | both |
| in, not in | Boolean | Infix | Verify if a value belongs to a set of values | 1 | YES | Standard |
| between | Boolean | Infix | Verify if a value belongs to a range of values | 1 | YES | both |
| isnull | Boolean | Functional | Compares the values with the NULL literal | 1 | YES | both |
| exists_in, not_exists_in, exists_in_all, not_exists_in_all | Boolean | Infix | Checks the Identifiers and the foreign keys | 2 | NO | Standard |
| match_characters | Boolean | Functional | Checks if a value respects or not a pattern | 1 | YES | Standard |
| all | Boolean | Functional | Verifies that all values in the Dataset are true | 1 | YES | Standard |
| any | Boolean | Functional | Verifies that at least one value in the | 1 | YES | Standard |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | Dataset are true | | | |
| unique | Boolean | Functional | Verifies that at only one value in the Dataset are true | 1 | YES | Standard |
| func_dep | Boolean | Functional | Checks the functional dependency between components of a Dataset | 1 | YES | Standard |
| and | Boolean | Infix | Calculates the logical AND | 2 | YES | both |
| or | Boolean | Infix | Calculates the logical OR | 2 | YES | both |
| xor | Boolean | Infix | Calculates the logical XOR | 2 | YES | both |
| not | Boolean | Infix | Calculates the logical NOT | 1 | YES | both |
| extract | Date operator | Functional | Returns an integer that is part of a given date | 1 | YES | Standard |
| string_from_date | Date operator | Functional | Converts a date value into a string | 1 | YES | Standard |
| current_date | Date operator | Functional | returns the current date and time | 0 | YES | Standard |
| union | Set | Functional | Computes the union of datasets | 1..N | NO | Standard |
| intersect | Set | Infix | Computes the intersection of datasets | 1..N | NO | Standard |
| symdiff | Set | Functional | Computes the symmetric difference of 2 datasets | 2 | NO | Standard |
| setdiff | Set | Infix | Computes the difference of 2 datasets | 2 | NO | Standard |

| | | | | | | |
|---|---|---|---|---|---|---|
| subscript | Set | Postfix | Assigns a fixed value to the identifires and remove them | 1 | NO | Standard |
| transcode | Set | Functional | Recodes the identifiers values using a mapping ruleset | 1 | YES | Standard |
| aggregate | Set | Functional | Aggregates data using a hierarchical ruleset. | 1 | YES | Standard |
| aggregateFunctions | Statistical function | Functional | Set of statistical functions used to aggregate data | 1 | YES | Standard |
| time_aggregate | Statistical function | Functional | Set of statistical functions used to aggregate data using time constraints | 1 | YES | Standard |
| analytic function | Statistical function | Functional | Allows to specify operations on groups of Data Points | 1 | YES | Standard |
| hierarchy | Statistical function | Functional | Applies a hierarchical aggregation | 1 | YES | Standard |
| check (with datapoint ruleset) | Validation | Functional | Applies one or more datapoint Ruleset on a Dataset. | 1 | YES | Standard |
| check (with hierarchical ruleset) | Validation | Functional | Applies one or more hierarchical ruleset on a Dataset. | 1 | YES | Standard |
| check (single rule) | Validation | Functional | Checks if an expression verifies a condition | 1 | NO | Standard |
| check value domain subset | Validation | Functional | Checks if the Value Domain Subset is respected | 1 | NO | Standard |
| fill_time_series | Time series | Functional | Replaces each missing data point in the input Dataset | 1 | YES | Standard |

| | | | | | | |
|---|---|---|---|---|---|---|
| flow_to_stock | Time series | Functional | Transforms from a flow interpretation of a Dataset to stock | 1 | YES | Standard |
| stock_to_flow | Time series | Functional | transforms from a stock interpretation of a Dataset to flow | 1 | YES | Standard |
| timeshift | Time series | Functional | Shifts the time component of a specified range of time | 1 | YES | Standard |
| if-then-else | Conditional | Functional | Makes different calculations according to a condition | 1 | YES | Standard |
| nvl | Conditional | Functional | Replaces the null value with a value | 1 | YES | Standard |
| rename | Clause | Clause (Postfix Operator) | change the name and the role of Measures or Attributes component | 1 | YES (ONLY) | Standard |
| filter | Clause | Clause (Postfix Operator) | Filters the Data Points | 1 | YES (ONLY) | Standard |
| keep | Clause | Clause (Postfix Operator) | Alters the Data Structure | 1 | YES (ONLY) | Standard |
| calc | Clause | Clause (Postfix Operator) | Calcuates the values of a Structure Component | 1 | YES (ONLY) | Standard |
| attrcalc | Clause | Clause (Postfix Operator) | Calculates the values of an Attribute | 1 | YES (ONLY) | Standard |

# VTL-ML - Evaluation order of the Operators

Within a single expression of the manipulation language, the operators are applied in sequence, according to the precedence order. Operators with the same precedence level are applied according to associativity rules. Precedence and associativity orders are reported in the following table.

| Order | Operator | Description | Associativity |
|---|---|---|---|
| I | () | Round parenthesis. To alter the default order. | Left-to-right |
| II | All VTL functional operators | The majority of the operators of the VTL | Left-to-right |
| III | Clauses and membership | | Left-to-right |
| IV | unary plus<br>unary minus<br>not | Unary minus<br>Unary plus<br>Logical negation | Right-to-left |
| V | *,<br>/ | Multiplication<br>Division | Left-to-right |
| VI | +, - | Addition<br>Subtraction | Left-to-right |
| VII | > >=<br>< <=<br>in, not in<br>between | Greater than<br>Less than<br>In (not in) a value list<br>In a range | Left-to-right |
| VIII | exists_in<br>not_exists_in<br>exists_in_all<br>not_exists_in_all | Identifiers matching | Left-to-right |
| IX | =<br><> | Equal-to<br>Not-equal-to | Left-to-right |
| X | and | Logical AND | Left-to-right |
| XI | or<br>xor | Logical OR<br>Logical XOR | Left-to-right |
| XII | if-then-else | Conditional (if-then-else) | Right-to-left |
| XIII | := | Assignment | Right-to-left |

# Syntactical conventions

In the remainder of the document, and in the Syntax sections in particular, a meta-syntax is used to describe the syntax of the operators. The meta-syntax is described in this section and is **not part** of the VTL language, but has only presentation purposes.

- *For denoting the type of a Variable Parameter, we refer to the* ***– VTL types*** **(See User Manual, Section "Objects and Types")***.*
- *Operator names and parameters are* ***case sensitive****.*
- *In general, some operators have infix style, others have functional style and the clauses have postfix style.*

The syntax of the operators is defined by *meta-expressions*, which denotes the signature of an operator, that is, its **name**, the list of **the input parameters**, the possible special **keywords** and the respective **types.** For readability reasons, a meta-expression is often partitioned into concatenated sub-meta-expressions (or simply sub-expressions), as follows:

*meta-expression ::= sub-expr1 sub-expr2 … sub-exprN*
*sub-expr1 ::= sub-meta-expression …*
*…*
*sub-exprN ::= sub-meta-expression …*

In this representation:

- The *sub-expr1, … sub-exprN* are meta-variables, that is, placeholders for sub-expressions. In the text, they are in italic.
- The symbol *::==* means "defined as" and denotes the assignment of a sub-expression to a meta-variable.
- The operator names and the special keywords that appear in the various sub-expressions are in **bold**.

Sub-expressions can be composed into the meta-expression adopting a particular restriction of **regular expression patterns** as follows:

- *{optional}, {optional}?,[optional]? : alternative ways to denote an optional sub-expression*
- *{one-or-more}+: a sub-expression that is repeated from 1 to many occurrences*
- *{zero-or-more}*: a sub-expression that is repeated from 0 to many occurrences*
- *[part1|part2|part3]: alternative sub-expressions*
- *[part1|part2|part3]+: alternative sub-expressions, from 1 to many occurrences*
- *[part1|part2|part3]*: alternative sub-expressions, from 0 to many occurrences*

Example

**[trim | ltrim | rtrim** ] **(** *ds* **)**

ds : dataset {identifier <IDENT> as scalar-type}+
{measure <IDENT> as string-literal}+
{attribute <IDENT> as scalar-type}*

The meta-expression above synthesizes:

- **trim, ltrim, rtrim**, “**(**“, “**)**” are the operator names (reserved keywords);
- They take s input an expression ds, which is a meta-sub-expression and defined accordingly;
- the type of ds is constrained to be a *Dataset* with one or more Identifier Components and one or more string Measure Component. No particular constraints are introduced for attributes.
- *ds* is the only parameter of the operators in the example and denotes a Dataset. Specifically, <IDENT> is a placeholder for any identifier (measure or attribute, in the different cases).

From this template, it is possible to infer some valid instances of the operators:

ds_1 **:= ltrim**(ds_2)
ds_1 **:= rtrim**(ds_3)

The two examples above are compliant with the template. In facts, **ltrim** and **rtrim** are recognized as VTL operators of the library and ds_2 and ds_3 are two Datasets. Also observe that the example implies a previous definition of ds_2 and ds_3, for example importing the Datasets from the database (as we will see, with the GET operator). The restrictions on the specific structure of the input Datasets, in terms of allowed Identifier and Measure Components, are also checked, but do not have effects on the syntax.

# VTL-DL - Artefacts Definition

## defineValueDomain

*Semantics*

The operator **defineValueDomain** defines a ValueDomain in the VTL information model.

*Syntax*

**defineValueDomain** valueDomainId **(**
{*valueDomainDescription***,** *isEnumerated*}
*dimensionType* { [ *inLineCodeList* | *dataTypeRestriction*] }
**)**

*Parameters*

*valueDomainId* : ident
*valueDomainDescription* : string
*isEnumeraed* : boolean
*dimensionType* : scalar
*inLineCodeList* : **list(** { **record({**@codeItemId **as** ident; { #codeDescr **as** constant; } **})** }* **)**
*codeItemId* : ident
*codeDescr* : constant
*dataTypeRestriction*
: **restrict** [**YYYY** | **MM** | **DD** | **YYYY-MM** | **maxLength** n | **regexp** regexp | **between** a **and** b | **>** b | **<** n | **<=** n | **>=** n]
*n, a, b* **:** numeric
*regexp* : string

- *valueDomainId* – is the identifier of the new ValueDomain.
- *valueDomainDescription* – is a string that describes the new ValueDomain.
- *isEnumerated* – is a Boolean that denotes whether the new ValueDomain is enumerated.
- *dimensionType* – is the data type of the Identifier Component.
- *inLineCodeList* – is an in line specification of a CodeList. It is a list of records (pairs, in particular). The first element of the record is the *codeItemId* (which identifies the code item, is the identifier of the record "@"), the second, optional, is the codeDescription, that is, the actual value for the code item. An *in-Line CodeList* cannot be reused.
- *regexp* – is a regular expression.
- *dataTypeRestriction* – constrains the allowed values by restricting *dimensionType*.

*Constraints*

- The scalar-type of the constant codeDescr must be dimensionType.
- *regexp* is a POSIX regular expression.
- If the ValueDomain is enumerated, an inLineCodeList must be specified.
- The particular restriction for *dataTypeRestriction* must be coherent with *dimensionDataType*. In particular:
    - date: [**YYYY** | **MM** | **DD** | **YYYY-MM** | **> YYYY-MM-DD** | **< YYYY-MM-DD** | **>=** …]
    - string: maxLength n, regexp regexp
    - number: [between a and b | <a | >a | …]

*Returns*

This operator defines persistent ValueDomain artefacts that can be referenced by a reference to *valueDomainId*. References valueDomainRef to *valueDomainId* are implicitly created in the VTL information model.

*Semantic specification*

This operator takes as input an identifier for this ValueDomain and the specification for its dimension. The dimension is in turn a component ValueDomain.
Only mono-dimensional ValueDomains can be defined, the multi-dimensional ValueDomain are implicitly defined in the VTL information model as the Cartesian product of the mono-dimensional ones. The allowed

values are directly those specified by the criteria. Otherwise, in case of n-dimensional ValueDomains, all the combinations of values of the mono-dimensional ValueDomains are possible, which means that the ValueDomain contains the Cartesian product of the values of the single mono-dimensional ValueDomain.
The definition of a ValueDomain comports also the implicitly definition of the respective full ValueDomainSubset, that is the subset of the ValueDomain that allows the same values of the ValueDomain, without further restrictions to its domain.
The dimension defines a set of allowed values by means of one among different criteria: 1) with an in-line definition of a codeList; 2) by restricting the *dimensionType* to a subset of allowed values by means of a criterion out of a set of pre-defined ones; 3) by allowing all the values of the specified *dimensionType*.
Notice that a CodeList can only be defined within a *ValueDomain* or a *ValueDomainSubset*, using the in-line mode.
After the application of the operator, the information model is modified as follows.
A *ValueDomain* identified by *valueDomainId* is created. Its description *ValueDomainDescr* is set to the value of *valueDomainDescription*, when specified, NULL otherwise. Property isEnumerated is set according to parameter *isEnumerated*. Property DataType is set to *dimensionType*.
Anytime a ValueDomain is specified, are implicitly specified with him all the possible combination between the new ValueDomain and the others. Therefore, the definition of a ValueDomain defined also multi-dimensional ValueDomain. In addition, a ValueDomainSubset full is defined and its ValueDomainRef is set to *valueDomainId*.
For the dimension, a corresponding **component** ValueDomain identified by the respective *compValueDomainId* is created in the information model and its properties are set as follows. The property valueDomainDescription contains the string "component ValueDomainSubset of *id*", where *id* amounts to *valueDomainId*. Property DataType is set to *dimensionType*.

*Examples*
1) This example defines the ValueDomain TimeYears as a restriction of the date type where only the digits representing the years are considered.
**define** ValueDomain TimeYears **(**"Time values", date **restrict YYYY)**

2) This example defines the ValueDomain GeoAreas with an in-line CodeList, that is the enumeration of all the allowed values.
**define** ValueDomain GeoAreas**(**"Geographic areas", string list(record(@IT, "Italy"), record(@LU, "Luxembourg"),""**,...))**

# defineValueDomainSubset

*Semantics*
The operator **defineValueDomainSubset** defines a ValueDomainSubset in the VTL information model.

*Syntax*
**defineValueDomainSubset** *valueDomainSubsetId* **(**
{*valueDomainSubsetDescription*, *isEnumerated*}
*valueDomainRef* { [ *inLineSubCodeList* | *dataTypeRestriction*] }
**)**

*Parameters*
*valueDomainSubsetId* : ident
*valueDomainSubsetDescription* : string
*isEnumberated : Boolean*
*valueDomainRef* : valueDomain-ref
*inLineSubCodeList*: **list(** { **record({@**codeItemId **as** ident; { **#**codeDescr **as** constant; } **})** }* **)**
*dataTypeRestriction*
: **restrict** [**YYYY** | **MM** | **DD** | **YYYY-MM** | **maxLength** n | **regexp** regexp | **between** a **and** b | **>** b | **<** n | **<=** n | **>=** n]
*n, a, b* **:** numeric
*regexp* : string

- *valueDomainSubsetId* – is the identifier of the new ValueDomainSubset.
- *valueDomainSubsetDescription* – is a string that describes the new ValueDomainSubset.

- *isEnumerated* – specifies whether the ValueDomainSubset is enumerated.
- *valueDomainRef* – is the reference to an existing ValueDomain.
- inLineCodeList – is an in line specification of a CodeList. It is a list of records (pairs, in particular). The first *element* of the record is the codeItemId (which identifies the code item, is the identifier of the record "@"), the second, optional, is the codeDescription, that is, the actual value for the code item. An in-line CodeList cannot be reused.
- *regexp* –is a POSIX regular expression.
- *dataTypeRestriction* – constrains the allowed values by restricting dimensionType of the referred ValueDomain.

### *Constraints*

- *regexp* is a POSIX regular expression.
- The possible restrictions on the values of the dimension must be coherent with the type of the dimension in the *ValueDomain* referred to by *valueDomainRef*.
- The criteria according to which the values of the dimension is defined must be the same as in the referred *ValueDomain*, that is: 1) if an *in-Line CodeList* is used in the *ValueDomain*, then in the *ValueDomainSubset* an *in-Line CodeList* containing a subset of the values must be used; 2) if a *dataTypeRestriction* has been used in the *ValueDomain*, then a *dataTypeRestriction* must be used in the *ValueDomainSubset*.
- If the ValueDomainSubset is enumerated, an inLineSubCodeList must be specified.
- Independently of the way in which the values of the dimension are defined, the allowed values for the dimension of the *ValueDomainSubset* must be a subset of the allowed values in the referred *ValueDomain* for the respective dimension.
- The particular restriction for *dataTypeRestriction* must be coherent with *dimensionDataType* of the referred *ValueDomain*. In particular:
    - date : [**YYYY | MM | DD | YYYY-MM | > YYYY-MM-DD | < YYYY-MM-DD | >=** …]
    - string: maxLength n, regexp
    - number: [**between** a **and** b | <a | >a | …]

### *Returns*

This operator defines persistent *ValueDomainSubset* artefacts that can be referenced by a reference to *valueDomainSubsetId*. References *valueDomainRef* to *valueDomainId* are implicitly created in the VTL information model.

### *Semantic specification*

This operator takes as input an identifier for this ValueDomainSubset, a reference to an existing ValueDomain and the specification for its dimension in terms of subsets of the dimension of the referred ValueDomain.
If no further constraints are posed, all the values that are allowed in the dimension of the ValueDomain are allowed in the ValueDomainSubset as well; alternatively, restrictions on the dimension can be specified according to a set of criteria.
Only mono-dimensional ValueDomainSubsets can be defined, the multi-dimensional ValueDomainSubset are implicitly defined in the VTL information model as the Cartesian product of the mono-dimensional ones. The allowed values are directly those specified by the criteria. Otherwise, in case of n-dimensional ValueDomainSubsets, all the combinations of values of the mono-dimensional ValueDomainSubsets are possible, which means that the ValueDomainSubset contains the Cartesian product of the values of the single mono-dimensional ValueDomainSubset.
The general rule is that the restrictions for the dimension must produce a subset of the values that are present in the ValueDomain for that dimension.
The allowed criteria are the following: 1) with an in-line definition of a sub CodeList; 2) by restricting the dimensionType to a subset of allowed values by means of a criterion out of a set of pre-defined ones; 3) by allowing all the values that are allowed in the referred ValueDomain.
If in the ValueDomain no restriction is applied, in the ValueDomainSubset any restriction that is coherent with the type of the respective dimension can be applied (hence no restriction, CodeList specification, data type restriction). If in the ValueDomain a CodeList (defining it in an in-line fashion) is specified, the ValueDomainSubset can either inherit all the values (no restriction) or restrict such CodeList specifying an in-line subset CodeList. If in the ValueDomain a *dataTypeRestriction* is adopted, the ValueDomainSubset can either inherit all the values (no restriction) or use another *dataTypeRestriction* that produces a subset of the parent one when applied to the original *dimensionType*.
After the application of the operator, the information model is modified as follows.
A ValueDomainSubset, identified by ValueDomainSubsetId, is created (the value of the property SetId is set to ValueDomainSubsetId). Its description SetDescr is set to the value of *valueDomainSubsetDescription*, if present,

NULL otherwise. Property isEnumerated is set according to parameter *isEnumerated* and coherently with the Propery SetCriterion. Property Criterion is set to "IN_LINE_CODELIST" (in-line CodeList), "RESTRICTION" (type restriction), "FULL" (all the values of the referenced dimension) depending on how the allowed values have been specified.
For the dimension, a corresponding **component** ValueDomainSubset is created and its properties are set as follows.
The identifier SetId is set to ValueDomainSubsetId concatenated to the string "_REF_" concatenated to the compValueDomainId of the referred component ValueDomain. The property SetDescr contains the string "component ValueDomainSubset of *id*", where *id* amounts to compValueDomainId of the referred component ValueDomain. Property SetCriterion is set to "IN_LINE_CODELIST" (in-line CodeList), "RESTRICTION" (type restriction), "FULL" (all the values of the referenced dimension) depending on how the allowed values have been specified.
Note that unlike in the ValueDomains, the identifiers for the component ValueDomainSubsets are statically specified and cannot be overridden. They are unique for a given ValueDomain and ValueDomainSubsets, so that a ValueDomain can be restricted in different ways. Moreover, note that there is no need for an artefact memorizing the relationship between the component ValueDomainSubsets and the compound ones, since it can be directly inferred from the identifiers conventions.
This operator also allows to alter existing ValueDomainSubset in a basic way. If a ValueDomainSubset with the same valueDomainSubsetId already exists in the information model, it is replaced by the newly defined one. The same holds for the respective component.

*Examples*
1) This example defines a ValueDomainSubset of positive numbers as a restriction of a ValueDomain allowing any integer number.
**define** ValueDomain Numbers("Integer Numbers", integer);
**define** ValueDomainSubset PositiveNumbers ("Number greater than 0", Numbers, **restrict** > 0)

2) This example defines a ValueDomainSubset for email addresses, as a restriction of a ValueDomain allowing any string.
**define** ValueDomain EmailAddresses("E-mail addresses", string-literal);
define ValueDomainSubset validEmailAddress("Valid e-mail addresses", EmailAddresses, restrict "[a-z]+@[a-z].[a-z]+"

# defineVariable

*Semantics*
The operator **defineVariable** defines a persistent Variable in the VTL information model.

*Syntax*
**defineVariable** *variableId*

*Parameters*
*variableId* : ident

# defineDataStructure

*Semantics*
The operator **defineDataStructure** defines a persistent DataStructure in the VTL information model

*Syntax*
**defineDataStructure** *dataStructureId* **(**
{*dataStructureDescr*}
{ [ *componentType* **(** *componentName* [**Identifier** | **Measure** | **Attribute**] **)** |
*valueDomainSubsetRef* **(** *componentName* [**Identifier** | **Measure** | **Attribute**] **)** ] **;** }+
**)**

*Parameters*

*dataStructureId* : ident
*dataStructureDescr* : string
*componentType* : scalar
*componentName* : ident
*valueDomainSubsetRef* : valueDomainSubset-ref

- *dataStructureId* – is the identifier of the new *DataStructure*.
- *dataStructureDescr* – is a string that describe the new *DataStructure*.
- *componentType* – is the type of a Component in the new *DataStructure*.
- *componentName* – is a string that represents the name of the Component in the new *DataStructure*.
- *valueDomainSubsetRef* – is a reference to an existing *ValueDomainSubset*, used to assign a specific type to a Component.

*Constraints*

- At least one IdentifierComponent must be defined.
- At least one MeasureComponent must be defined.
- There cannot be two components with the same *componentName*.

*Returns*

This operator defines a persistent DataStructure artefact that can be referenced by a reference to *dataStructureId*, in the VTL information model.

*Semantic specification*

This operator defines a persistent DataStructure in the information model, allowing to specify its name and the description, along with the characteristics of its Components. It takes as input the identifier for this DataStructure, according to the conventions for it, optionally a description, and the specification for one or more Components. The Components can be defined in two ways: in a simplified form where there is a *componentName*, and a scalar-type for it is directly specified (*componentType*); in a fully-fledged form, where there is a *componentName*, and a ValueDomainSubset (mono-dimensional) is specified to restrict the allowed values.

Although in the VTL information model, a Component is always characterized by a ValueDomainSubset, the simplified form is particularly useful, since it prevents the need to define a ValueDomain and a ValueDomainSubset that are the mere renaming of a scalar data type. Let us now consider the fully-fledged form.

The ValueDomainSubset is mono-dimensional, it restricts the allowed values for a single Component;

For each Component, a role must be declared by using one keyword among **Identifier**, **Measure** and **Attribute**.

After the application of the operator, the information model is modified as follows.

A DataStructure identified by *dataStructureId* is created. Its description DataStructureDescr is set to the value of *dataStructureDescr*, when specified, NULL otherwise.

For each Component, a DataStructureComponent is created. Its identifier, componentId, takes its value from the parameter *componentName*, which is unique within a single DataStructure. The DataStructureComponent is linked to the referred DataStructure by assigning the DataStructureId property.

If the Component is specified in the simplified form (only the data type), the created DataStructureComponent is linked (by the property SetId) to a conventional ValueDomainSubset for that type. Notice that a conventional ValueDomainSubset that simply renames each scalar type and the corresponding ValueDomain are assumed to be present in the information model, or created when needed and then reused.

If the DataStructureComponent is specified in the fully-fledged form (with its ValueDomainSubset), the single ValueDomainSubset is referred to by the property SetId.

In all the cases the property VariableRole is set to “Identifier”, “Measure” and “Attribute” depending on the used keyword.

For each component a new RepresentedVariable is created (or an existing one is reused). Its identifier, VariableId, is automatically and the respective property of DataStructureComponent is assigned accordingly. The description of the variable is automatically generated as “RepresentedVariable for <componentId>”. The RepresentedVariable is linked to the ValueDomain it takes its values from (being restricted by a specific ValueDomainSubset when assigned to a DataStructureComponent).

*Examples*

1) Definition of a DataStructure where scalar types are used.

```
define DataStructure dstr_1(
        string ID identifier;
```

string NAME **Identifier**;
integer AGE **Measure**;
)

2) The example below allows to define a data structure using a ValueDomainSubset:

**define** ValueDomain Numbers(“Integer Numbers”, integer);
**define** ValueDomainSubset PositiveNumbers (“Number greater than 0”, Numbers, **restrict** > 0)

**define DataStructure** dstr_1(
string ID **Identifier**;
string NAME **Measure**;
PositiveNumbers AGE **Measure**;
)

## defineDataset

*Semantics*

The operator **defineDataset** defines a persistent Dataset in the VTL information model.

*Syntax*

**defineDataset** *datasetId* **(**
{*datasetDescr*,} {**IsCollected**},
[ *dataStructureRef* |
{ [ *componentType* **(** *componentName* [**Identifier** | **Measure** | **Attribute**] **)** |
*valueDomainSubsetRef* **(** *componentName* [**Identifier** | **Measure** | **Attribute**] **)** ] **;** }+
]
**)**

*Parameters*

*datasetId* : ident
*datasetDescr* : string
*dataStructureRef* : dataStructure-ref
*componentType* : scalar-type
*componentName* : string
*valueDomainSubsetRef* : valueDomainSubset-ref

- *datasetId* - is the identifier of the new Dataset.
- *datasetDescr* – is a string that describes the new Dataset.
- *isCollected* – if present this Dataset is an elementary one, otherwise it is meant to be the result of a calculation.
- *dataStructureRef* – is a reference to an existing DataStructure, used to assign a specific structure to the new Dataset. Optionally the DataStructure for the new Dataset can be defined in-line.
- *componentType* – is the type of a Component in the new Dataset.
- *componentName* – is a string that represents the name of the Component in the new Dataset.
- *valueDomainSubsetRef* – is a reference to an existing *ValueDomainSubset*, used to assign a specific type to a Component.

*Constraints*

- At least one Identifier Component must be defined.
- At least one Measure Component must be defined.
- There cannot be two components with the same *componentName*.

*Returns*

This operator defines a persistent Dataset artefact that can be referenced by a reference to *datasetId*, in the VTL information model.

*Semantic specification*

This operator defines a persistent Dataset in the information model, allowing to specify its name and the description, along with the characteristics of its Components (either specifying an existing DataStructure or defining the Components in an in-line fashion mode).
It takes as input the identifier for this Dataset, according to the conventions for it, optionally a description, and the reference to an existent DataStructure or alternatively the specification for one or more Components. The Components can be defined in two ways: in a simplified form where there is a *componentName*, and a scalar-type for it is directly specified (*componentType*); in a fully-fledged form, where there is a *componentName*, and a ValueDomainSubset is specified to restrict the allowed values.
Although in the VTL information model, a Component is always characterized by a ValueDomainSubset, the simplified form is particularly useful, since it prevents the need to define a ValueDomain and a ValueDomainSubset that are the mere renaming of a scalar data type. Let us now consider the fully-fledged form.
In case of in-line definition of the Components: for each Component, a role must be declared by using one keyword among **Identifier**, **Measure** and **Attribute**.
After the application of the operator, the information model is modified as follows.
In case of reference to an existing DataStructure.
A Dataset identified by *datasetId* is created. Its description DatasetDescr is set to the value of *datasetDescr*, when specified, NULL otherwise. The Dataset is linked to the DataStructure referred using *dataStructureRef* by assigning the DataStructureId property of the DataStructure identifier to the DataStructureId property of the new Dataset.
In case of definition of a new DataStructure (not reusable).
A Dataset identified by *datasetId* is created. Its description DatasetDescr is set to the value of *datasetDescr*, when specified, NULL otherwise. A DataStructure identified by an auto-generated DataStructureId is created for the new Dataset) and linked to it by assigning the generated identifier to the DataStructureId property of the Dataset. The description of the DataStructure is also generated automatically and set to "dataStructure_of_datasetId_description" (if the DataStructure is reused, this convention for the description will be violated).
For each Component, a DataStructureComponent is created (or the ones in the existing DataStructure are reused). Its identifier, *componentId*, takes its value from the parameter *componentName* (notice that for a DataStructure to be reused, these identifiers must be coherent), which is unique within a single DataStructure.
The DataStructureComponent is linked to the referred DataStructure by assigning the DataStructureId property.
If the Component is specified in the simplified form (only the data type), the created DataStructureComponent is linked (by the property SetId) to a conventional ValueDomainSubset for that type. Notice that a conventional ValueDomainSubset that simply renames each scalar type and the corresponding ValueDomain are assumed to be present in the information model, or created when needed and then reused.
If the StructureComponent is specified in the fully-fledged form (with its ValueDomainSubset, the single ValueDomainSubset is referred to by the property SetId. The property VariableRole is set to "Identifier", "Measure" and "Attribute" depending on the used keyword.
For each component a new RepresentedVariable is created (or an existing one is reused). Its identifier, VariableId, is automatically and the respective property of DataStructureComponent is assigned accordingly. The description of the variable is automatically generated as "RepresentedVariable for <componenId>". The RepresentedVariable is linked to the ValueDomain it takes its values from (being restricted by a specific ValueDomainSubset when assigned to a StructureComponent).

*Examples*

1) Definition of a Dataset, using an existing DataStructure.
**define Dataset** d_1( "Dataset with the same structure of dstr_1", dstr_1)
dstr_1 is a DataStructure previously defined.

2) Definition of a Dataset with an in-line DataStructure definition where scalar types are used.
**define Dataset** d_1(
    string ID **identifier**;
    string NAME **Identifier**;
    integer AGE **Measure**;
)

# VTL-DL - Rulesets

## define datapoint ruleset

*Semantics*

**define datapoint ruleset** defines a persistent object that contains Rules to be applied to each individual Data Point of a given Dataset. These rulesets are also called "horizontal" taking into account the tabular representation of a Dataset (considered as a mathematical function), in which each (vertical) column represents a Variable and each (horizontal) row represents a Data Point: these rulesets are applied on individual Data Points (rows), i.e. horizontally on the tabular representation.

*Syntax*

**define datapoint ruleset** *rulesetId* **(** *RulesetSignature* **) is**
    { *Rule* } { ; *Rule*}*
**end datapoint ruleset**
*Rule*
    ::={ *ruleId*:} { **when** *antecedentCondition* **then** } *consequentCondition*
    { **errorcode (** *errorCode* **)** }
    { **errorlevel (** *errorLevel* **)** }
*RulesetSignature*
    ::= *variable-signature* {, *variable-signature*}*
*variable-signature*
    ::= *variable-ref* {**as** *constant-string*}?

*Parameters*

*rulesetId* : identifier
*ruleId* : identifier
*antecedentCondition* : Boolean-scalar-expression
*consequentCondition* : Boolean-scalar-expression
*errorCode* : string
*errorLevel*: integer-literal
*constant-string*: string

- *rulesetId* – the identifier of the datapoint ruleset to be defined.
- *rulesetSignature* – the signature of the Ruleset. It specifies the Represented Variables (see the information model) on which the Ruleset is defined.
- *variable-signature* – it specifies a single Represented Variable on which the Ruleset is defined
- *variable-ref* - the reference to a Variable on which the Ruleset is defined. The Variable name can be aliased for the sake of compactness in writing the Rules. If the alias is not specified, the complete name of the Variable must be used in the body of the rules.
- *rule* – the complete specification of a single rule, as defined in the following parameters.
- *ruleId* – the identifier of the specific rule within the Ruleset. The *ruleId* is optional and, if not specified, is assumed to be the progressive order number of the Rule in the Ruleset (please note that this practice may cause changes of the rule identifiers in case the Ruleset is maintained, e.g. if new rules are added or existing rules are deleted)
- *antecedentCondition* - a Boolean scalar expression. It can contain references to the Variables declared for this Ruleset and Constants. All the Component level operators are allowed.
- *consequentCondition* - a Boolean scalar expression. It is evaluated when the *antecedentCondition* evaluates to true (missing antecedent conditions are assumed as true). It can contain references to the Variables declared for this Ruleset and Constants. All the Component level operators are allowed.
- *errorCode* – a string denoting the error code associated to the rule, respecting VTL conventions, in case the rule is used for validation.
- *errorLevel* - an integer containing the error level (severity) associated to the rule, in case the rule is used for validation.

- *constant-string*: the name assigned to the Variable within the ruleset

### Constraints

- *antecedentCondition* and *consequentCondition* cannot use Variables that are not defined in the *RulesetSignature*
- A *Variable* can appear only once in the *RulesetSignature*
- Either the *ruleId* is specified for all the rules of the Ruleset or for none.
- If specified, the *ruleId* must be unique within the Ruleset.

### Returns

A persistent DataPoint Ruleset identified by *rulesetId*, which can be referenced and used both for validation and data filtering (within a filter clause) purposes.

### Semantic specification

A DataPoint Ruleset (also "horizontal ruleset") is a persistent object that contains Rules to be applied to the Data Points of a given Dataset[1]. When used for validation, the Rules are aimed at checking the combinations of values of the Data Point Variables, assessing if these values fulfill the conditions expressed by the Rules themselves. The Rules are evaluated independently for each data point, returning a Boolean scalar value (see the **check** operator and the relevant options). When used for data filtering, the Rules are aimed at filtering the Data Points, maintaining only the ones that fulfill (or, as an option, that do not fulfill) the Rules themselves (see the **filter** operator and the relevant options).
Each rule contains an *antecedentCondition* Boolean expression followed by a *consequentCondition* Boolean expression and expresses a logical implication. Each condition states that when the *antecedentCondition* evaluates to **true**, for a given Data Point, then the *consequentCondition* must evaluate to **true** as well. In case the *antecedentCondition* is absent then it is assumed to be always true, therefore the *consequentCondition* must evaluate to true for all the Data Points. See the example below:

| *Rule* | *Meaning* |
|---|---|
| **when** flow = "CREDIT" or "DEBIT" **then** obs_value >= 0 | When the Variable named "flow" takes the value "CREDIT" or the value "DEBIT", then the Variable named "obs_value" has a zero or positive value. |
| **when** flow = "BALANCE" **then** obs_value between -1.000.000 and +1.000.000 | When the Variable named "flow" takes the value "BALANCE, then the Variable named "obs_value" has a value between -1.000.000 and +1.000.000 |

The definition of a Ruleset comprises a **signature** (*RulesetSignature*), which specifies the Represented Variables on which the Ruleset is defined and a number of **rules**, that are the Boolean expressions to be applied for each Data Point. The Rules can refer only to the Variables of the Ruleset signature, and must refer to all of them (in either the *antecedentCondition* or the *consequentCondition*, or both).
In regard to the Information Model, the Variables of the Ruleset signature identify a multi-dimensional space (i.e. a multi-dimensional Represented Variable), while each Rule provides for a criterion that demarcates a Set of values belonging to this space (i.e. a Set of combinations of values of these Variables).
A Ruleset can be applied on any Dataset which includes, among its Structure Components, the Variables of the Ruleset signature. More Rulesets having different signatures may be applied on the same Dataset, provided that the previous condition is satisfied.

[1] In order to apply the Ruleset to more Datasets, these Datasets must be joined together using the appropriate VTL operators in order to obtain a single Dataset.

Rules are uniquely identified by a *ruleId*. When the Ruleset is used for validation, two new Variables (the RULESET and the RULE Variables) are added in the Dataset that contains the validation result and filled with *rulesetId* and *ruleId* respectively, in order to document to which rules the results are referred. If not explicitly declared, the *ruleId* is assumed by default to be the progressive order number of the Rule in the Ruleset (please note that using the default mechanism the Rules identifiers can change if the Ruleset is maintained, e.g. if new Rules are added or existing Rules are deleted, and therefore the users that interpret the validation results must be aware of these changes).
As said, every **rule** is applied in a row-wise fashion to each individual Data Point of a Dataset. The references to the Variables defined in the *antecedentCondition* and *consequentCondition* are replaced with the values of the respective Variables of the Data Point under evaluation.
The semantics of each rule is the typical logical implication:
*antecedentCondition and consequentCondition*
The rule evaluates to true if: *antecedentCondition* evaluates to FALSE or *consequentCondition* evaluates to TRUE. In practice, the *consequentCondition* must be evaluated only if the *antecedentCondition* succeeds and therefore the former can be also interpreted as the precondition to apply the latter.
In the case of validation, the outcome is a Dataset (the validation output) having a Boolean measure (TRUE or FALSE) and broken down at least by the Variables RULESET and RULE containing respectively the *rulesetId* and the *ruleId* of the applied rule (for more details see the **check** operator). The variables ERRORCODE and ERRORLEVEL are also added in the output Dataset and valued with the parameters *errorCode* and *errorLevel* of the applied Rule in case of validation failure (i.e. FALSE value as outcome of the Rule).
These Rulesets can be also used to filter Datasets. In particular, the **filter** operator can apply a Horizontal|DataPoint Ruleset to all the Data Points of the Dataset to be filtered. The result will be a new Dataset, having the same data structure as the input Dataset and containing only the Data Points for which the Rules of the Ruleset evaluates to TRUE or optionally to FALSE (for more details see the filter operator).

*Examples*

1) Input Dataset:

| ds_bop | | | | | |
|---|---|---|---|---|---|
| TIME | REF_AREA | PARTNER | FLOW | OBS_VALUE | OBS_STATUS |
| 2010 | EU25 | CA | AVERAGE | 20 | |
| 2010 | BG | CA | NET | 1 | |
| 2010 | RO | CA | NET | 1 | M |
| 2010 | EU27 | CA | CREDIT | 12 | C |

```
define datapoint ruleset ruleset_1 (FLOW as x, OBS_STATUS as y) (
        flow_dr : when x = “CREDIT” or x = “AVERAGE” then y <> “C” errorcode ( -XXXXX )
)
```

Meaning:
Once ruleset_1 is defined, it is usable to perform validations or apply filters.
2)
ds := **check(**ds_bop, ruleset_1, **with measures , only failures)**

| TIME | REF_AREA | PARTNER | FLOW | RULE_ID | OBS_VALUE | OBS_STATUS | ERRORCODE |
|---|---|---|---|---|---|---|---|
| 2010 | EU27 | CA | CREDIT | ruleset1_flow_dr | 12 | C | -XXXXX |

3)
ds := ds_bop[**filter** ruleset_1]

| TIME | REF_AREA | PARTNER | FLOW | OBS_VALUE | OBS_STATUS |
|---|---|---|---|---|---|
| 2010 | EU25 | CA | AVERAGE | 20 | |
| 2010 | BG | CA | NET | 1 | |
| 2010 | RO | CA | NET | 1 | M |

# define hierarchical ruleset

*Semantics*

**define hierarchical ruleset** defines a persistent object that contains Rules to be applied to individual Components of a given Dataset in order to make validations or calculations according to hierarchical relationships between the relevant Code Items. These rulesets are also called "vertical" taking into account the tabular representation of a Data Set (considered as a mathematical function), in which each (vertical) column represents a Variable and each (horizontal) row represents a Data Point: these Rulesets are applied on Variables (columns), i.e. vertically on the tabular representation of a Data Set.

A first purpose of these Rules is to express some more aggregated Code Items (e.g. the continents) in terms of less aggregated ones (e.g., their countries). This kind of relations can be applied to aggregate data, for example to calculate an additive measure (e.g., the population) for the aggregated Code Items (e.g. the continents) as the sum of the corresponding measures of the less aggregated ones (e.g. their countries). If a certain information is available for both, the more and the less aggregated Code Items, these rules can be used for validating their mutual coherence, for example to check if the additive measures relevant to the aggregated Code Items (e.g. the continents) match the sum of the corresponding measures of their component Code Items (e.g. their countries).

Another purpose of these Rules is to express the relationships in which a Code Item represents some part of another one, (e.g., "Africa" and "Five largest countries of Africa", being the latter a detail of the former). This kind of relationships can be used only for validation, for example to check if a positive and additive measure (e.g. the population) relevant to the more aggregated Code Item (e.g., Africa) is greater than the corresponding measure of the other one more detailed (e.g. "5 largest countries of Africa").

The name "hierarchical" comes from the fact that this kind of Ruleset is able to express the hierarchical relationships between Code Items at different levels of detail, in which each (aggregated) Code Item is expressed as a partition of (disaggregated) ones.

As a first simple example, the following Hierarchical Ruleset named "BeneluxCountries" contains a single rule that asserts that, in the Value Domain "Geo_Area", the Code Item BENELUX is the aggregation of the Code Items BELGIUM, LUXEMBOURG and NETHERLANDS:

**define hierarchical ruleset** BeneluxCountriesHierarchy **( ValueDomain=**Geo_Area **)** *is*
BENELUX = BELGIUM + LUXEMBOURG + NETHERLANDS
**end hierarchical ruleset**

*Syntax*

**define hierarchical ruleset** *rulesetId* **(***RulesetSignature***)** *is*
{ Rule } { ; Rule}*
**end hierarchical ruleset**

*RulesetSignature*
*::= { antecedentSignature,} codeItemRelationSignature*
*antecedentSignature*
*::=* **antecedentvariables=** *variable-signature* {, *variable-signature*}*
*variable-signature*
::= *variable-ref* {**as** *constant-string*}?
*codeItemRelationSignature*
::= [**variable=** *variable-ref* | **valuedomain=** *valuedomain-ref*]
*Rule*
::= { ruleId **:** }? { **when** *antecedentCondition* **then** }? *codeItemRelation*
{ **errorcode (** *errorCode* **)** }?
{ **errorlevel (** *errorLevel* **)** }?
*antecedentCondition*
::= *boolean-scalar-expression*
*codeItemRelation*
::= *codeItem-reference* [ **=** | **>** | **<** | **>=** | **<=**] [ **+** | **-** ]? *codeItemReference* { [ **+** | **-** ] *codeItemReference*}*
*codeItemReference*
*::= codeItem-ref* [**from** *time-ref*]? [**to** *time-ref*]?

*Parameters*

*rulesetId* : identifier
*ruleId* : identifier

*codeItem-ref* : identifier
*variable-ref :* identifier
*valuedomain-ref :* identifier
*antecedentCondition* : boolean-scalar-expression
*errorCode* : string
*errorLevel* : integer
*time-ref* : time-literal

- *rulesetId* – the identifier of the Hierarchical Ruleset to be defined.
- *rulesetSignature* – the signature of the Ruleset. It specifies the space on which the Ruleset is defined.
- *antecedentSignature* - the signature of the antecedent conditions of the Ruleset. It specifies the Represented Variables (see the information model) on which the antecedent conditions of the Rules are defined.
- *codeItemRelationSignature* - the signature of the Code Item Relations of the Ruleset. It specifies either the Represented Variable or the ValueDomain (see the information model) on which the Code Item Relations of the Rules are defined. When a Represented Variable is specified, the Ruleset is meant to be applicable to DataSets having such Variable as a Component. When a Value Domain is specified, the Ruleset is meant to be applicable to Datasets having a Component which takes values on it.
- *variable-signature* – It specifies a single Represented Variable on which the Ruleset is defined
- *variable-ref* – It references a Represented Variable by its name. The Variable name can be aliased for the sake of compactness in writing the Rules. If the alias is not specified, the complete name of the Variable must be used in the body of the Rules.
- *constant-string*: the name assigned to the Variable within the ruleset
- *valueDomain-ref* – It specifies a Value Domain
- *Rule* – the complete specification of a single rule, as defined in the following parameters.
- *ruleId* – the identifier of the specific rule within the Ruleset. The ruleId is optional and, if not specified, is assumed to be the progressive order number of the Rule in the Ruleset (please note that this practice may cause changes of the rule identifiers in case the Ruleset is maintained, e.g. if new rules are added or existing rules are deleted)
- *antecedentCondition* – a Boolean scalar expression. All the Component level operators are allowed.
- *CodeItemRelation* – the specification of a Code Item Relation to be evaluated only when the *antecedentCondition* evaluates to true (missing antecedent conditions are assumed as true). It expresses a logical relationship between Code Items belonging to the Value Domain referenced by the Ruleset. The relation is expressed by one of the symbols “=”, “>”, “>=”, “<”, “<=”, which in this case denote special logical relationships typical of Code Items (see below). The first member of the relationship is a single Code Item. The second member of the relationship is the composition of one or more Code Items expressed by the symbols “+” or “-“, which in their turn also denote special logical operators typical of Code Items (see below). The meaning of these symbols is explained below.
- *codeItemReference* –the reference to an existing *Code Item* of the VTL information model, that is a Value of a ValueDomain.
- *errorCode* – a string denoting the error code associated to the rule, respecting VTL conventions, in case the rule is used for validation.
- *errorLevel* – an integer containing the error level (severity) associated to the rule, in case the rule is used for validation.

*Constraints*
- *valueDomainReference* must be enumerated.
- *antecedentCondition* must refer only to identifiers specified in *antecedentConditionIds*
- *errorCode* must respect the conventions of user-defined error codes.

*Returns*
A persistent Hierarchical (or vertical) Ruleset identified by *rulesetId*, which can be referenced and used both for validation and aggregation purposes.

*Semantic specification*
This operator defines a Hierarchical Ruleset, which is a collection of Rules expressing logical relationships between the Values (Code Items) of a Variable or a Value Domain.
Each rule contains an optional antecedent condition, which defines the cases in which the Rule has to be applied (if not declared the Rule is applied ever) and a mandatory code item relation, which expresses the **relation**

**between Code Items** to be enforced. In the relation, one Code Item (the first member of the relation) is put in relation to a combination of other Code Items.
As for the mathematical meaning of the relation, please note that each Value (Code Item) is the representation of an event belonging to a space of events (i.e. the relevant Value Domain), according to the notions of "event" and "space of events" of the probability theory (see also the section on the Generic Models for Variables and Value Domains in the VTL IM). Therefore the relations between Values (Code Items) express logical implications between events.
The envisaged types of relations are: "coincides" (=), "implies" (<), "implies or coincides" (<=), "is implied by" (>), "is implied by or coincides" (>=)[2]. For example:

UnitedKingdom < Europe means UnitedKingdom implies Europe

In other words, this means that if a point of space belongs to United Kingdom it also belongs to Europe.

January 2000 < year 2000 means January of the year 2000 implies the year 2000

In other word, if a time instant belong to "January 2000" it also belongs to the "year 2000"

The first member of a Relation is a single Code Item. The second member can be either a single code item, like in the example above, or a **logical composition of Code Items** giving another Code Item as result. The logical composition can be defined by means of Code Item Operators, whose goal is to compose some Code Items in order to obtain another Code Item.
Please note that the symbols "+" and "-" do not denote the usual operations of sum and subtraction, but logical operations between Code Items which are seen as events of the probability theory. In other words, two or more Code Items cannot be summed or subtracted to obtain another Code Item, because they are events and not numbers, however they can be manipulated through logical operations like "OR" and "Complement".
Note also that the "+" also acts as a declaration that all the Code Items denoted by "+" in the formula are mutually exclusive one another (i.e. the corresponding events cannot happen at the same time), as well as the "-" acts as a declaration that all the Code Items denoted by "-" in the formula are mutually exclusive one another and furthermore that each one of them is a part of (implies) the result of the composition of all the Code Items having the "+" sign.
At intuitive level, the symbol "+" means "*with*" (Benelux = Belgium *with* Luxembourg *with* Netherland) while the symbol "-" means "*without*" (EUwithoutUK = EuropeanUnion *without* UnitedKingdom).
When these relationships are applied to additive numeric measures (e.g. the population relevant to geographical areas), they allow to obtain the measure values of the compound Code Items (i.e. the population of Benelux and EUwithoutUK) by summing or subtracting the measure values relevant to the component Code Items (i.e. the population of Belgium, Luxembourg and Netherland in the former case, EuropeanUnion and UnitedKingdom in the latter). This is why these logical operations are denoted in VTL through the same symbols as the usual sum and subtraction. Please note also that this is valid whichever is the Data Set and whichever is the additive measure (provided that the possible other Identifier Components of the Data Set Structure have the same values), so that the Rulesets of this kind are potentially reusable.
The Ruleset Signature specifies the space on which the Ruleset is defined. The "antecedentSignature" specifies the Variables on which the antecedent conditions of the Rules are defined (the Rules can refer only to these Variables and must refer to all of them). The "codeItemRelationSignature" specifies either the Represented Variable or the ValueDomain (see the information model) on which the Code Item Relations can be defined (when a Represented Variable is specified, the Ruleset is meant to be applicable to DataSets having such Variable as a Component, when a Value Domain is specified, the Ruleset is meant to be applicable to Datasets having a Component which takes values on it).
The Hiererchical Ruleset may act on one or more Measures of the input Data Set provided that these measures are additive (for example it cannot be applied on a measure containing a "mean" because it is not additive).

If a **Hierarchical Ruleset** is **used for calculation** (see also the "Calc" operator), only the Relations expressing coincidence ("=") are evaluated (provided that the *antecedentCondition* is true). The result Data Set will contain the compound Code Items (the left members of those relations) calculated from the component Code Items (the right member of those Relations). Moreover, the clauses typical of the validation are ignored (e.g. ErrorCode, ErrorLevel/Severity).
If some Code Items are defined equal to themselves, the relevant Data Points are brought in the result unchanged. For example, the following Ruleset will maintain in the result the Data Points of the input Data Set relevant to Belgium, Luxembourg and Netherland and will add new Data Points containing the calculated value for Benelux:

**define hierarchical ruleset** *AddBenelux* **(valuedomain=***GeoArea***)** *is*
Belgium = Belgium
Luxembourg = Luxembourg

[2] "Coincides" means "implies and is implied"

```
        Netherlands = Netherlands
        Benelux = Belgium + Luxembourg + Netherlands
    end hierarchical ruleset
```

If a **Hierarchical Ruleset** is **used for validation** (see also the "Check" operators for more detailed information), all the possible Relations ("=", ">", ">=","<","<=") are evaluated (provided that the *antecedentCondition* is true). The Rules are evaluated independently. The Data Points referred both by the left and the right members of the Relations are taken from the input Dataset. The Antecedent Condition is evaluated and, if "TRUE", the Code Item Relation is also evaluated (the operations specified in the right member of the Relation are performed and the result is compared to the first member according to the specified Relation). The possible relations in which Code Items are defined as equal to themselves are ignored. The result Data Set will contain, as a Measure, the Boolean result of the validation, and, as Identifiers, the RulesetId, the RuleId and the Identifiers of the input Data Set. The possible clauses typical of the validation are applied (e.g. ErrorCode, ErrorLevel/Severity) and generate additional Measures in the result. Further options are better explained in the Check operator).
Please note again that in case of validation the Data Points relevant to both the members of the Relations are expected to belong to the input Data Set. As obvious, if the data to be validated are originally in different DataSets, either they can be merged in advance using other VTL operators or the validation can be done by comparing those Data Sets directly (see also the Check operator), without using this kind of Ruleset.

**The Hierarchical Rulesets allow to declare the time validity of Rules and Relations**. Firstly, the Antecedent Condition may be referred to a time variable, expressing when the Code Item Relation has to be applied (i.e. when it is considered valid as a whole). Secondly, each Code Item of the second member of the Code Item Relation can be qualified with a time validity, so expressing when the Code Item participates in the relation. As a default, when not expressed the validity is considered to be "ever".

The following two simplified examples show possible ways of defining the European Union in term of Countries.

<u>Example 1</u>

```
define hierarchical ruleset  EuroAreaCountries1 (antecedentvariable=Time,  variable=GeoArea) is
        when Time between 1.1.1958 and 31.12.1972
                then EU = BE + FR + DE + IT + LU + NL
        when Time between 1.1.1973 and 31.12.1980
                then EU = … same as above … + DK + IE + GB
        when Time between 1.1.1981 and 02.10.1985
                then EU = … same as above … + GR
        when Time between 1.1.1986 and 31.12.1994
                then EU = … same as above … + ES + PT
        when Time between 1.1.1995 and 30.04.2004
                then EU = … same as above … + AT + FI + SE
        when Time between 1.5.2004 and 31.12.2006
                then EU = … same as above … + CY + CZ + EE + HU + LT + LV + MT + PL + SI + SK
        when Time between 1.1.2007 and 30.06.2013
                then EU = … same as above … + BG + RO
        when Time >= 1.7.2013
                then EU = … same as above … + HR
end hierarchical ruleset
```

<u>Example 2</u>

```
define hierarchical ruleset  EuroAreaCountries2 (valuedomain=Geo_Area) is
        EU = AT (from 1.1.1995) + BE (from 1.1.1958) + BG (from 1.1.2007)
             +  …  +  GB (from 1.1.1973) +  …
             + SE (from 1.1.1995) + SI (from 1.5.2004) + SK (from 1.5.2004)
end hierarchical ruleset
```

In this example, when GB will exit from UE, the GB term would become:
+ GB (from 1.1.1973 to … the future date of Brexit …)

**The Hierarchical Rulesets allow defining hierarchies** either having or not having levels (free hierarchies). For example, leaving aside the time validity for sake of simplicity:

```
define hierarchical ruleset  GeoHierarchy (valuedomain=Geo_Area) is
        World = Africa + America + Asia + Europe + Oceania
```

Africa = Algeria + … + Zimbabwe
America = Argentina + … + Venezuela
Asia = Afghanistan + … + Yemen
Europe = Albania + … + Vatican City
Oceania = Australia + … + Vanuatu
Afghanistan = AF_reg_01 + … + AF_reg_N
… … … … … …
Zimbabwe = ZW_reg_01 + … + ZW_reg_M
EuropeanUnion = … + … + … + …
CentralAmericaCommonMarket = … + … + … + …
OECD_Area = … + … + … + …
end hierarchical **ruleset**

Hierarchies may be useful for validation in case more levels of detail are contained in the Data Set to be validated. The Hierarchical Rulesets defines the mutual coherency rules of these different levels of detail. Because the various Rules can be evaluated independently, their order is not significant.

Hierarchies may also be useful for calculations. For example, they can be used to calculate the upper levels of the hierarchy if the data relevant to the leafs (or some other intermediate level) are available in the input Data Set. For example, having additive measures broken by region, it would be possible to calculate these measures broken by countries, continents and the world. Besides, having additive measures broken by country, it would be possible to calculate the same measures broken by continents and the world.

In the Hierarchies there can be dependencies between Rules, because the inputs of some Rules can be the output of other Rules, so the former can be evaluated only after the latter. For example, the data relevant to the Continents can be calculated only after the calculation of the data relevant to the Countries. As a consequence, the order of calculation of the Rules is determined by their mutual dependencies and can be different from the order of the Rules in the Ruleset. The dependencies between the Rules form a directed acyclic graph.

**Hierarchical Rulesets allow defining multiple relations for the same Code Item**.
Multiple relations are often useful for validation. For example, the Balance of Payments item "Transport" can be broken down both by type of carrier (Air transport, Sea transport, Land transport) and by type of objects transported (Passengers and Freights) and both breakdowns must sum up to the whole "Transport" figure. In the following example a RuleId is assigned to the different methods of breaking down the Transport.

**define hierarchical ruleset** *TransportBreakdown* **(valuedomain=** BoPItem **)** *is*
transport_method1 : Transport = AirTransport + SeaTransport + LandTransport,
transport_method2: Transport = PassengersTransport + FreightsTransport
**end hierarchical ruleset**

Multiple relations can be deemed as useful even in some case of calculation. For example, imagine that the input Data Set contains data about resident units broken down by region and data about non-residents units broken down by country. In order to calculate an homogeneous level of aggregation (e.g. by country), a possible Ruleset might be the following:

**define hierarchical ruleset** CalcCountryLevel **( valuedomain=**Geo_Area **)** *is*
Country1 = Country1
Country1 = Region11 + … + Region1M
…
CountryN = CountryN
CountryN = Region N1 + … + RegionNM
**end hierarchical ruleset**

A warning is opportune about the possible practice of calculating the same Code Item in more Rules (calculation methods) of the same Ruleset. The Rulesets of this kind, in fact, may produce either right or wrong figures depending on the content of the input Data Set.
As a matter of fact, in the calculation the outcomes of all the Rules belonging to the Ruleset are aggregated together to produce the final result, in order to remove possible duplicates in the Identifiers (duplicate values in the Identifiers cannot be allowed, see also the Information Model). As far as each Code Item is defined just once as left member of a relation, the values of the Identifiers of the results of the single Rules are all distinct and their aggregation cannot generate inconsistencies. This is not ever true if a Code Item is defined more than once (e.g. through more than one calculation method).
In the Ruleset of the example above, each Country is calculated using two calculation methods, whose results may have the same keys, which will be aggregated together. The output Data Set will be correct provided that, in the input Data Set, any information is present either by country or by region (never both of them). The output

Data Set would contain errors if some information is present in the input Data Set both by country and by region: the resulting figures would be indicatively (and wrongly) doubled.
In general, if more left members refer to the same Code Item (in other words, if a Code Item is calculated through more calculation methods), the result may be inconsistent for some input DataSets. It is possible to avoid these situations by using other approaches for calculating the desired result (e.g. splitting the Ruleset, calculating the result in more steps, using antecedentConditions, using other VTL operators). This example has been presented to better clarify the behavior of this kind of Ruleset and warn about possible limitations to its reusability.

*Examples*

1) The Code Item Relation is defined on the Variable “sex”: Total is defined as Male + Female.
No antecedent conditions are defined.

```
define hierarchical ruleset vr_sex (Variable= sex) is
    TOTAL = MALE + FEMALE;
end hierarchical ruleset
```

2) BENELUX is the aggregation of the Code Items BELGIUM, LUXEMBOURG and NETHERLANDS, if not true, the errorcode is 2000 and the errorlevel is high (10)

```
define hierarchical ruleset BeneluxCountriesHierarchy ( valuedomain=Geo_Area) is
BENELUX = BELGIUM + LUXEMBOURG + NETHERLANDS errorcode 2000 errorlevel 10
end hierarchical ruleset
```

3) American economic partners. The first rule verifies that the value reported for North America is greater than the value reported for US. This type of validation is useful when the data communicated by the data provider do not cover the whole composition of the aggregate but only the main elements. No antecedent conditions are defined.

```
define hierarchical ruleset vr_american_partners (variable= counterpart_area) is
    NORTH_AMERICA > US ;
    SOUTH_AMERICA = BR + UY + AR + CL ;
end hierarchical ruleset
```

4) Example of item having multiple definitions. The Balance of Payments item "Transport" can be broken down by type of carrier (Air transport, Sea transport, Land transport) and by type of objects transported (Passengers and Freights) and both breakdowns must sum up to the total "Transport" figure.

```
define hierarchical ruleset vr_bop (variable= bop_item ) is
    transport_method1 : Transport = AirTransport + SeaTransport + LandTransport,
    transport_method2 : Transport = PassengersTransport + FreightsTransport
end hierarchical ruleset
```

## define mapping ruleset

*Semantics*

The **define mapping** allows to transcode a set of values of an Identifier Component

*Syntax*

```
define mapping map_1 (
        { condition ( IdentifierComponent<?> idCond {, IdentifierComponent<?> idCond } * ) }
        map_to ( IdentifierComponent<?> idMapTo )
        map_from ( IdentifierComponent<?> idMapFrom )
) is
        { MappingRule ; } +

MappingRule:= { when Component<Boolean> whenCondition then }
                        IdentifierValue valueTo = IdentifierValue valueFrom
end mapping ruleset
```

*Parameters*

*idCond* : identifier
*idMapTo* : component-ref
*idMapFrom* : component-ref
*whenCondition* : boolean
*valueTo* : string
*valueFrom* : string

- *idCond* is the identifier used in the condition part. More than one identifier can be used.
- *idMapTo* is the identifier whose values are resulting from the conversion of values of *idMapFrom*
- *idMapFrom* is the identifier whose values are converted to values of *idMapTo*
- *whenCondition* is a boolean expression. When *whenCondition* is evaluated to true then the corresponding mapping rule is executed. If *whenCondition* is omitted in a rule then it is implicitly assumed to be true.
- *valueTo* is a valid value for *idMapTo*
- *valueFrom* is a valid value for *idMapFrom*

*Constraints*

*idCond*, *idMapFrom* and *idMapTo* are the names of existing Identifier Components). *valueTo* is a valid value for *idMapTo* and *valueFrom* is a valid value for *idMapFrom*.

*Semantic specification*

It creates a mapping that can be applied to transcode a set of values using the **transcode** statement. A mapping is a set of rules for transcoding values belonging to the code lists of two identifier components.

*Returns*

None.

*Examples*

See the examples under the **transcode** operator.

# VTL-ML - General purpose operators and functions

## Parentheses ( )

*Semantics*

The parenthesis allows to modify the default order of evaluation of the operators.

*Syntax*

( *expression* )

*Constraints*

None.

*Semantic specification*

## Assignment :=

*Semantics*

The ":=" symbol allows to assign the value of an expression to a variable parameter.

*Syntax*

*variable_parameter* **:=** *expression*

*Constraints*

None.

*Semantic specification*

the *expression* may evaluate to any data type.

*Examples*

Assignment of a Constant<Number> value to a parameter:

numpi **:=** 3.14

Assignment of a String value to a parameter:

str **:=** "hello world"

Assignment of an expression to a parameter:

popA **:=** populationDS + 1

Assignment of a Dataset expression to a parameter:

ds_1 **:=** get("NAMESPACE/DF_NAME/2000.USD.M.F.A.BOP.ANN.STO.EABL")

Assignment of a Constant<Boolean> value to a parameter:

bool_var **:=** true

## Membership .

*Semantics*

The membership operator allows to specify a single component of a Dataset

*Syntax*

*ds* **.** *comp*

*Parameters*

*ds* : Dataset

*comp* : Dataset component-ref

- *ds* – is a Dataset
- *comp* – a valid component of ds

*Constraints*
*None.*

*Returns*
A Dataset having all the identifiers and only one Measure or Attribute c specified by the operator.

*Semantic specification*
The membership operator is particularly useful to work with operators that have specific constraints in terms of the types of the Measure Components or have more than one Measure Component.

*Examples*
1) Suppose ds_1 is a multi-measure Dataset, where M1 is a numeric Measure Component and M2 is a string Measure Component, let ds_2 be a mono-measure Dataset with a single Measure Component M1. ds_1 and ds_2 have the same Identifier Components. Let us supposed the sum ds_1 + ds_2 is desired.
The following syntax: ds_1.M1 + ds_2 represents the resulting Dataset. In this notation ds_1 is temporarily considered mono-measure

2) Suppose the comparison operator ("=") needs to be applied on the Component COUNTRY of the Dataset ds_1. In this expression:
ds_2 := ds_1.COUNTRY="Luxembourg"
the membership operator specifies that the Identifier Component COUNTRY is temporarily considered as the only Measure Component to be used in the comparison.

3) Suppose it is needed to round an Component. The round operator acts on Measure Components, which must be all Numeric. Suppose we have a Dataset ds_1 with a string Measure Component DESCRIPTION and a numeric Component AVERAGE_AGE, which needs to be rounded to the 3rd decimal. The expression:
ds_1 := round(ds_1.AVERAGE_AGE,3)
performs this task.
ds_1.AVERAGE_AGE temporarily considers AVERAGE_AGE the only numeric Measure Component of ds_1. The round is then normally applied.

4) Let us suppose we have two multi-measure Datasets ds_1 and ds_2, having the same Identifier Components K1 and K2, and the same Measure Components M1 (which is a Numeric), M2 which is a String.
The expression:
ds_3 := ds_1.M1 + ds_2.M1
sums only the Measure Component M1.

5) Let us suppose we have two multi-measure Datasets ds_1 and ds_2, having the same Identifier Components K1 and K2, and the Measure Components M1 and M2.
The expression:
ds_3 := ds_1.M1 + ds_2.M2
sums the Measure Component M1 with the measure component M2.

## Alias as

*Semantics*
The **as** operator allows to rename one component of a Dataset or the component resulting by an expression.

*Syntax*
*ds.comp **as** alias*

*Parameters*
*ds* : Dataset
*alias* : string

*Constraints*

This operator works only on Measure components.

*Semantics specification*

The operator takes as input a Dataset and the identifier of a Measure Component, and returns a new Dataset having only that Measure Component and all the original Identifier Components.

*Examples*

1) Let us suppose we have two multi-measure Datasets ds_1 and ds_2, having the same Identifier Components K1 and K2, and the Measure Components M1and M2.

The expression:

ds_3 := ds_1.M1 + ds_2.M2 as “M1”

sums the Measure Component M1 with the measure component M2. The outcome Dataset has one Measure Components: M1, which is obtained as the sum of M1 in ds_1 and M2 in ds_2.

2) Let us suppose we have a Datasets ds_1 and the Measure Components M1.

The expression:

ds_2 := ds_1.M1* 10 as “M2”

returns a Dataset having only one measure components M2 obtained as the product of M1 and 10.

## alterDataset

*Semantics*

The **alterDataset** allows to maintain all or a subset of components of the input Dataset having the identifier role.

*Syntax*

**alterDataset(** *ds_1{, compList}* { **all** } **);**

*Parameters*

*ds_1* : dataset {identifier <IDENT> as scalar-type}+{measure <IDENT> as scalar-type}*
{attribute <IDENT> as scalar-type}*

*compList* : list<list<component-ref>>

- *ds_1* – is the Dataset that the operator uses to produce the resulting Dataset.
- *compList*– is the set of components belonging to the input Dataset.
- ***all*** – its definition implies the presence of all components of *ds_1* in the resulting Dataset.

*Constraints*

None.

*Returns*

This operator returns a Dataset having only Identifiers Components. The components of the returned Dataset are all the components of the input Dataset that are part of the *compList* or, if it is not specified, only the identifier components of the input Dataset. If one or more measures or attributes are included in the list, they will be part of the returned Dataset but having a role of identifiers. This operator allows removing identifier components from the input Dataset removing duplications.

*Semantic specification*

The Dataset resulting will have only Identifiers also if it contains components that were previously measures. If the **with measures** flag is specified then the resulting Set will have as added Identifiers, the Measures Components of the input one Dataset, too.

*Examples*

1) **alterDataset(**ds_1 **all)**

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 100 |
| 2 | B | 200 |

| set_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 100 |
| 2 | B | 200 |

2) l_1 = list<components-ref> (REF_AREA)

**alterDataset(**ds_1,l1**)**

| IT_nord_pop | | |
|---|---|---|
| **TIME** | **REF_AREA** | **OBS_VALUE** |
| 2015 | ITCD | 27799803 |
| 2015 | ITC | 16138643 |
| 2015 | ITC1 | 4424467 |
| 2015 | ITC2 | 128298 |
| 2015 | ITC3 | 1583263 |
| 2015 | ITC4 | 10002615 |
| 2015 | ITD | 11661160 |
| 2015 | ITD1 | 518518 |
| 2015 | ITD2 | 537416 |
| 2015 | ITD3 | 4927596 |
| 2015 | ITD4 | 1227122 |
| 2015 | ITD5 | 4450508 |
| 2014 | ITCD | 27785211 |
| 2014 | ITC | 16130725 |
| 2014 | ITC1 | 4436798 |
| 2014 | ITC2 | 128591 |
| 2014 | ITC3 | 1591939 |
| 2014 | ITC4 | 9973397 |
| 2014 | ITD | 11654486 |
| 2014 | ITD1 | 515714 |
| 2014 | ITD2 | 536237 |
| 2014 | ITD3 | 4926818 |
| 2014 | ITD4 | 1229363 |
| 2014 | ITD5 | 4446354 |

| set_1 |
| --- |
| **REF_AREA** |
| ITCD |
| ITC |
| ITC1 |
| ITC2 |
| ITC3 |
| ITC4 |
| ITD |
| ITD1 |
| ITD2 |
| ITD3 |
| ITD4 |
| ITD5 |

# get

*Semantics*

The **get** operator allows to fetch all the instances of a Dataset from the system and returns a Dataset containing them.

*Syntax*

**get(**
    *ds_id* {, *ds_id*}*
    {,**keep(**keepPart {, keepPart }***)**}
    {,**dedup**(*consResFunctions*)}
    {,**filter(***filterPart***)**}
    {,**aggregate( aggregateFunction (**aggrPart {, aggrPart}***)***)**}
**)**

*Parameters*

*ds_id* : ident
*consResFunctions* : list<component-ref * (t*t) -> t > (t is the type of the referred Component)
*keepPart* : component-ref
*filterPart* : boolean
*aggrPart* : component-ref (Component<Numeric>)

- *ds_id* – is the Persistent Dataset to be fetched.
- *keepPart* – is a valid reference to a Component of ds_id.
- *consResFunctions* – is a List of reference to valid Components of ds and conflict resolution Function.
- *filterPart* – is a boolean Component expression which is evaluated row-wise and states if a row is to be kept (if evaluates to true) or removed (if it does not evaluate to true) from the result.
- *aggrPart* – is a valid reference to the numeric Measure Component to aggregate.

*Constraints*

- All the input Datasets ds_id must be persistent (see put operator) and must have the same Logical Data Structure, which is the same Components in number, name and type (static).
- If more than a Dataset ds_id is defined, then the definition of consResFunctions is mandatory.
- The consResFunction List, must defines a conflict resolution function for each Measure Component specified in the keep clause. For each Component the respective conflict resolution function must return a value of the

same type (as explained in the syntax). If consResFunction is not used and duplicated records are present the get operator return an error.

- keepPart must be a Component expression containing exactly the name of a Component of any ds (complex Component expressions, combining more than one Component are not allowed) (static).
- aggrPart must be a Component expression containing exactly the name of a Measure Component present in any ds (no complex Component expressions, combining more than one Component is allowed). If there is at least one aggrPart, there must be one for each Measure Component that is present in a keepPart. If keepPart is omitted, all Measure Components must be in the aggregate. This means that there cannot be Measure Components, kept that are not used in aggregations (static).

*Returns*

A Dataset obtained as the union of all the Datasets specified by the identifiers *ds*, keeping only the columns specified in the *keepParts* and the rows in the *filterParts*, choosing from duplicate Datapoints through *consResFunctions*, aggregating over all the Measure Components in *aggrParts* grouping.

*Semantic specification*

The operator **get**, is the data retrieval command. It takes in input a number (at least one) of Dataset *ds*. Together with *put*, it is the only operator in VTL where a persistent Dataset can be mentioned.
The command operates as follows: considers all the instances of the identified Dataset (selected according to the semantics of the identifier); builds a union without duplicates (conflicts are resolved using the *consResFunctions* specified in the dedup part); keeps in the result only the Components that are present in the *keepPart* (like SQL SELECT). If the keep part is omitted, all the Components are preserved in the result; selects the only instances returning *true* for the *filterPart* boolean Component expression. For the *filterPart*, any complex boolean Component expression over all the Datasets Components (not only the ones mentioned in the *keepPart* can be used) and it is evaluated row-wise (like SQL WHERE).
Finally, the command aggregates (like SQL aggregations and GROUP BY) applying an aggregation function (see aggregate function operator) over the Measure Component specified in *aggrPart* grouping by the Identifier Components that are kept in the *keepPart* (or all if there is no *keepPart*).
NULL values are considered in aggregations only if the "*include NULLS*" part is present. Specifically, they propagate as usual resulting in a NULL sum, average or median if at least one NULL is present among the values; in a NULL minimum or maximum if the only value to aggregate coincides with NULL; they are considered as always distinct in both count and count_distinct.
Viceversa, if *include* NULLS part is absent, NULL values are not considered in aggregations.

*Examples*

1) The expression:
ds_1 := **get**("*DF_NAME/*2000-2010.USD.M.F.A.BOP.ANN.STO", keep(K1, K2, M1))

Retrieves Dataset identified by *DF_NAME/*2000-2010.USD.M.F.A.BOP.ANN.STO from the system, keeping the Identifier Components K1 and K2 and the Measure Component M1.

| DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 5 |
| 2 | B | 7 |

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 5 |
| 2 | B | 7 |

2) The expression:
ds_1 := **get**("DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO", "DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO", keep(K1, K2, K3, M1), dedup(M1*min))

Retrieves the union of Datasets identified by DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO and DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO, keeping the Identifier Components K1, K2 and K3 and the Measure Component M1 for all of them.

| DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO | | | |
|---|---|---|---|
| **K1** | **K2** | **K3** | **M1** |
| 1 | A | X | 5 |
| 2 | B | Y | 7 |

| DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO | | | |
|---|---|---|---|
| **K1** | **K2** | **K3** | **M1** |
| 1 | A | X | 6 |
| 2 | B | Y | 7 |
| 3 | C | Z | 9 |

| ds_1 | | | |
|---|---|---|---|
| **K1** | **K2** | **K3** | **M1** |
| 1 | A | X | 5 |
| 2 | B | Y | 7 |
| 3 | C | Z | 9 |

The union had produced two duplicates: (1,A,X,5) and (1,A,X,6), (2,B,Y,7) and (2,B,Y,7). The min conflict resolution function take care of the minimum value for M1 between the duplicates.

3) The expression:

ds_1 := **get**("DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO", ("DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO", keep(K1, K2, K3, M1), dedup(M1*min), filter(K3="X")))

retrieves the union of Datasets identified by DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO and DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO keeping the Identifier Components K1, K2 and K3 and the Measure Component M1 for all of them and selecting only the rows where the value of the Component K3 equals to the Constant<String> "X".

| NAMESPACE/DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO | | | |
|---|---|---|---|
| **K1** | **K2** | **K3** | **M1** |
| 1 | A | X | 5 |
| 2 | B | Y | 7 |

| NAMESPACE/DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO | | | |
|---|---|---|---|
| **K1** | **K2** | **K3** | **M1** |
| 1 | A | X | 6 |

| | | | |
|---|---|---|---|
| 2 | B | Y | 7 |
| 3 | C | Z | 9 |

| ds_1 | | | |
|---|---|---|---|
| **K1** | **K2** | **K3** | **M1** |
| 1 | A | X | 5 |

4)

The expression:

ds_1 := get(
"DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO", "DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO",
keep(K1, K2, M1), dedup(M1*min), aggregate(sum(M1)))

retrieves the union of Datasets identified by

DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO

and

DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO

keeping the Identifier Components K1 and K2 and the Measure Component M1 for all of them. It aggregates over the Measure Component M1, grouping by the Identifier Components K1 and K2.

| DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO | | | | |
|---|---|---|---|---|
| **K1** | **K2** | **K3** | **M1** | **M2** |
| 1 | A | X | 5 | 2 |
| 2 | B | Y | 7 | 3 |

| DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO | | | | |
|---|---|---|---|---|
| **K1** | **K2** | **K3** | **M1** | **M2** |
| 1 | A | Y | 6 | 5 |
| 2 | B | Y | 7 | 7 |
| 3 | C | Z | 9 | 11 |

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 11 |
| 2 | B | 14 |
| 3 | C | 9 |

5) The expression:

ds_1 := **get**("DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO", “DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO",
keep(K1, K2, M1,M2), dedup(M1*min, M2*first_value), filter(K3>5 or K3=1), aggregate(sum(M1),max(M2)))

retrieves the union of Datasets identified by DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO and DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO keeping the Identifier Components K1 and K2 and the Measure Components M1 and M2 for all of them. It selects only the rows where K3 is greater than 5 or exactly 1. It aggregates over the Measure Component M1 by sum, over the Measure Component M2 by max, grouping by the Identifier Components K1 and K2.

| DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO | | | | |
|---|---|---|---|---|
| **K1** | **K2** | **K3** | **M1** | **M2** |
| 1 | A | 10 | 5 | 2 |
| 2 | B | 1 | 7 | 3 |

| DF_NAME/2011-2012.USD.M.F.A.BOP.ANN.STO | | | | |
|---|---|---|---|---|
| **K1** | **K2** | **K3** | **M1** | **M2** |
| 1 | A | 25 | 6 | 5 |
| 2 | B | 1 | 7 | 7 |
| 3 | C | 3 | 9 | 11 |

| ds_1 | | | |
|---|---|---|---|
| **K1** | **K2** | **M1** | **M2** |
| 1 | A | 11 | 5 |
| 2 | B | 14 | 7 |

## put

*Semantics*

It stores the content of a Dataset expression ds into a persistent Dataset.

*Syntax*

**put(***ds*, *ds_id***)**

*Parameters*

*ds, ds_id* : dataset {identifier <IDENT> as scalar-type}+
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*

- *ds* – is the Dataset, or Dataset expression which contents must be stored in the system.
- *ds_id* – is the Dataset that will assumes the contents of ds, it will be persistent in the system.

*Constraints*

The Logical Data Structure of *ds* must conform to the one of the Dataset in the system that is identified by *ds_id* (static).

*Returns*

A Dataset that is a copy of the input one *ds.*

*Examples*

1) The expression below is to store the ds_1 Dataset.

ds := put(ds_1, "DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO")

2) ds := put(log((ds_1 + ds_2),10), "DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO")
The result of logarithm is stored, while the sum is not persistent.

3) ds := put(log(put(ds_1 + ds_2, "DF_NAME/2000-2011.USD.M.F.A.BOP.ANN.STO"),10),"DF_NAME/2000-2010.USD.M.F.A.BOP.ANN.STO")
Both the results of the sum and the logarithm are stored into the system. The fact that put outputs the input expression allows for this kind of use.

# eval

*Semantics*

The **eval** operator allows to execute an external, non-VTL program, and returns its result as a Dataset.

*Syntax*

**eval (**Constant<String> *language*,
[{**script=**}Constant<String> *script* | Constant<String> *programPath*],
{,{**params=**}ConstantList<?> parameterList}
, {**dataset=**}PersistentDataset *ds_id***)**

*Parameters*

*ds_id* : dataset {identifier <IDENT> as scalar-type}+
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*
*language*: string
*script*: string
*programPath* : string
*parameterList* : list<scalar-type>

*ds_id* – the PersistentDataset the program saves into.
*language* – is the programming language of the script.
*script* – is the code of the script.
*programPath* – a path to a script file.
*parameterList* – the List of input parameters for the script.

*Constraints*

- language must be the name of a programming language, meaningful and executable in the target system (such as a SQL stored-procedure language, R, STATA, etc.) (dynamic).
- script must be the code of a program, valid with respect to the specified language. The program can perform whatever internal logic, but is forced to calculate and autonomously store exactly one PersistentDataset, ds_id (dynamic).
- programPath must be a valid path in the target system to a program file, compliant with the specified language. It does not necessarily correspond to a filesystem file, but can also be the identifier of a DBMS stored procedure, and so forth (dynamic).
- parameterList must be compatible in order and type with the input parameters of the script (dynamic).

*Semantic specification*

The program specified in the eval operator, is user-defined and can perform any internal logic, however it has to adhere to some conventions:

- it can take as input only String or Numeric parameters, which are directly bound to parameterList;
- it must autonomously store its results into a single Dataset ds_id. Indeed, the operator fetches the saved Dataset (like a common get operation) and returns it as output, which can be handled within other VTL expressions;
- it must calculate exactly one Dataset;
- it cannot refer to a parameter variable, but can only work with physical objects, such as relational tables (for SQL), data frames (for R), which are loaded autonomously by the program with the appropriate commands. Therefore, if a Dataset that has been calculated in a previous step needs to be used within a user-defined program, it must be stored (with a put) into the system and loaded appropriately by the program logic afterwards;
- it must return 0 if it has terminated correctly, a negative number otherwise.

# Join expression

*Semantics*

The join expression implements some of the features of the FLWOR expression described in the VTL User Manual.

*Syntax*

{ **[** join_clause **]** } { body }

join_clause ::= { [ **<u>inner</u>** | **outer** | **cross** ] } { *ds* { **,** *ds* * } **on** *dim* { **,** *dim* } * }
*body* := **{** clause { **,** clause } * **}**
*clause* := *calc_clause* | *drop_clause* | *filter_clause* | *keep_clause* | *rename_clause* | *unfold_clause* | *fold_clause*
*calc_clause* := { role } *compName* **=** *k*
*drop_clause* ::= **drop** { cmp { **,** cmp } * }
*keep_clause* ::= **keep** { cmp { **,** cmp } * }
*filter_clause* ::= **filter** boolean-expression | dpr
*rename_clause* ::= **rename** cmp **to** cmp **{ ,** cmp **to** cmp }
*unfold_clause* ::= **unfold** dim **,** msr **to** elem { **,** elem }
*fold_clause* ::= **fold** elem { **,** elem } **to** dim **,** msr
*role* := **identifier** | **measure** | **attribute**

### *Parameters*

*ds* : [ dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type }*
{attribute <IDENT> as scalar-type} * ]
*cmp* : Component
*dim* : IdentifierComponent
*dpr* : name of a data point ruleset

*ds* – is a Dataset name
*alias* – is an alias for a Dataset, to be used when the same Dataset appears several times in
*k* – is a scalar expression, or a Dataset expression denoting a single measure or attribute
*role* – is the role of the calculated component *c*. If omitted then the role is derived from *k* (if k is a Dataset expression) otherwise the default role of *c* is **attribute**.
*dim* – is an Identifier Componentcommon to all Datasets specified in the join clause

### *Constraints*

For inner and outer joins (see below), one of the Datasets specified in the *join_clause* must contain all Identifier Components from all other Datasets from the join_clause (with the same name and the basic scalar type, number, boolean, string, or date) .
The name of the component cannot be **filter**, **keep** or **rename**, or those names must be quoted within "'".
A Dataset ds should appear only once in the list of Datasets.

### *Returns*

The Dataset returned by the last statement of the body.

### *Semantic specification*

This operator implements some of the features of the FLWOR expression described in the VTL manual part 1. Only the features that are useful for validation and transformation purposes are retained in the VTL operator.
First VTL executes the join clause and then the body.
The statements are executed in the specified order and operate on an input working Dataset.
The Dataset resulting from the join clause is the input for the first statement of the body.
The Dataset resulting from a statement is the input for the following statement.
The Dataset returned by the last statement of the body is returned as the final result of the join expression.

## join_clause

The meaning of the **inner** and the **outer** join is the same as the meaning of INNER JOIN and FULL OUTER JOIN constructs, respectively, in the SQL-92 standard. These are the differences:

| | |
|---|---|
| **inner** ds1, ds2 | the resulting Dataset contains the data points that exist both in ds1 and ds2 (i.e. the common Identifier Components of ds1 and ds2 have the same values in ds1 and ds2). |
| **outer** ds1, ds2 | the resulting Dataset contains the data points that exist either in ds1 or ds2. Measures and attributes of data points that exist only in ds1 or ds2 (but not in both) have the **null** value. |
| **cross** ds1, ds2 | the resulting Dataset contains all data points of ds1 combined with all data points of ds2 (i.e. the Cartesian product of ds1 and ds2). The statements contained in the body are expected to reduce the number of data points by filtering them as needed. Measures and attributes of data points that exist only in ds1 or ds2 (but not in both) have the **null** value. |

The join clause builds the input Dataset of the first statement, according to the following rules.

- If the join clause contains a single *Dataset* then that *Dataset* is the initial working *Dataset*. It is possible to refer to the components of the *Dataset* simply by using their name. Suppose that ds1 has a measure m, then

  **[** ds1 **]** **{** a = m +1 **}** correct
- For **inner, outer** and **cross** joins, the initial working *Dataset* is the result of the inner or outer or cross join applied to the *Dataset*s specified in the join clause. If the *Dataset*s have common measures or attributes (i.e. with identical names) then it is mandatory to refer to those components by specifying both the *Dataset* name and the measure name. Suppose that ds1 and ds2 have a common measure m, then:

  **[**ds1,ds2**]** **{** a = ds1.m +1 **}** correct

  **[**ds1,ds2**]** **{** a = m + 1 **}** not correct (ambiguous: m can refer to ds1 or ds2)

  The measures can be renamed with the rename clause:

  **[**ds1,ds2**]** **{ rename** 'ds1.m' to m1 , a = m1 + 1 **}** correct

  The use of the quotation is necessary because ds1.m syntactically is not a valid name (this exception to the syntax rules is allowed only in the join body).

  In the final result of the join expression the common measures and attributes that have not been renamed are automatically dropped. The same applies when the working Dataset is the input for a filter that uses a datapoint (horizontal) ruleset.
- If the **on** clause is specified then the join is possibly defined on a subset of the common Identifier Components of the *Dataset*s. If the *Dataset*s have common Identifier Components (i.e. with identical names, data type and values domain) that are not specified in the **on** clause then it is mandatory to refer to those Identifier Components by specifying both the *Dataset* name and the measure name. For example, if ds1 and ds2 have some common Identifier Components d1, d2 and d3, the following expression:

  **[** ds1,ds2 **on** d1, d2 **]**

  returns a Dataset with the following Identifier Components:

  d1, d2, 'ds1.d3', 'ds2.d3'

  the Identifier Components can be renamed using the rename clause:

  **[** ds1,ds2 **on** d1, d2 **]** **{ rename** 'ds1.d3' **to** new1, 'ds2_d3' **to** new2 **}**

  In the final result of join expression the common Identifier Components that are not listed in the **on** clause and have not been renamed are automatically renamed by replacing the "." with an underscore "_". The same applies when the working Dataset is the input for a filter that uses a datapoint (horizontal) ruleset.
- The join clause can be omitted. In this case VTL implicitly adds a join clause containing all *Dataset*s that are used inside the body. For example, the following join expression:

  **{** a = ds1.m1 + ds2.m1 **}**

  is automatically treated by VTL as equivalent to:

  **[inner** ds1, ds2**]** **{** a = ds1.m1 + ds2.m1 **}**

  and

  **[ outer ]** **{** a = ds1.m1 + ds2.m1 **}**

  is equivalent to:

  **[outer** ds1, ds2**]** **{** a = ds1.m1 + ds2.m1 **}**

*Examples*

1) inner join returns data points that exists in both Datasets

ds3 := **[** ds1, ds2 **]** **{**
obs_value = ds1.obs_value + ds2.obs_value ,
obs_status = ds1.obs_status
**}**

| ds1 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 20 | E |
| 2010 | BG | CA | 2 | P |
| 2010 | RO | CA | 2 | P |

| ds2 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 10 | P |

| ds3 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 30 | E |

the example above can be expressed equivalently as:

```
ds_bop3 := {
        obs_value = ds1.obs_value + ds2.obs_value ,
        obs_status = ds1.obs_status
        }
```

2) outer join returns data points that exist in at least one Dataset when a data point does not exist in the other Dataset, the value of its measures and components is null compare with the following example:

```
ds_bop3 := [outer ds1, ds2 ] {
        obs_value = ds1.obs_value + ds2.obs_value ,
        obs_status = ds1.obs_status
        }
```

| ds3 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 30 | E |
| 2010 | BG | CA | | P |
| 2010 | RO | CA | | P |

3)
nvl is used to replace the null value with 0 (compare with the previous example)

```
ds_bop3 := [outer ds1, ds2 ] {
        obs_value = ds1.obs_value + nvl (ds2.obs_value, 0) ,
        obs_status = ds1.obs_status
        }
```

| ds3 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 30 | E |
| 2010 | BG | CA | 2 | P |
| 2010 | RO | CA | 2 | P |

4)
example of join defined on a subset of the Identifier Components (family_id)

| ds_census | | | |
|---|---|---|---|
| PERSON_ID | FAMILY_ID | REL | NATIONALITY |
| 1 | 1 | HEAD | IT |
| 2 | 1 | SPOUSE | IT |
| 3 | 1 | CHILD | IT |

| | | | |
|---|---|---|---|
| 4 | 2 | HEAD | US |
| 5 | 2 | SPOUSE | US |
| 6 | 2 | CHILD | IT |
| 7 | 2 | CHILD | IT |

```
head := ds_census (rel=HEAD);
spouse := ds_census (rel=SPOUSE);
child := ds_census(rel= CHILD ) ;
[ head, spouse, child on family_id ] {
    rename head.person_id to head_id, spouse.person_id to spouse_id, child.person_id to child_id ;
    rename head. nationality to head_nationality, spouse. nationality to spouse_nationality, child. nationality to
    child_nationality ;
}
```

| ds_result | | | | | | |
|---|---|---|---|---|---|---|
| FAMILY_ID | HEAD_ID | SPOUSE_ID | CHILD_ID | HEAD_NATIONALITY | SPOUSE_NATIONALITY | CHILD_NATIONALITY |
| 1 | 1 | 2 | 3 | IT | IT | IT |
| 2 | 4 | 5 | 6 | US | US | IT |
| 2 | 4 | 5 | 7 | US | US | IT |

## calc_clause

calc_clause := { role } *compName* **=** *k*

The calc_clause adds a new component (Identifier , Measure or Attribute Component) or replaces an existing component (Measure or Attribute: the Identifier Components cannot be replaced) of the working Dataset. If *calc_comp* coincides with the name of an existing Component in the working Dataset (even with different type), the calculated one replaces the former, in name, value and type.

*Examples*

Suppose merge_flags is a user defined function (not shown here) that returns EP when applied to E, P

```
ds3 := [ ds1, ds2 ] {
            obs_value = ds1.obs_value + ds2.obs_value ,
            obs_status = merge_flags ( ds1.obs_status, ds2.obs_status )
          }
```

| ds3 | | | | |
|---|---|---|---|---|
| TIME | REF_AREA | PARTNER | OBS_VALUE | OBS_STATUS |
| 2010 | EU25 | CA | 30 | EP |

## drop_clause

drop_clause ::= **drop** { cmp { **,** cmp } * }

The **drop** clause drops from the working Dataset the measures and attributes specified.

*Examples*

ds2 := **[** ds1**]** **{** **drop** obs_status **}**

| ds1 | | | |
|---|---|---|---|
| TIME | REF_AREA | PARTNER | OBS_VALUE |
| 2010 | EU25 | CA | 20 |
| 2010 | BG | CA | 2 |

| | | | |
|---|---|---|---|
| 2010 | RO | CA | 2 |

## keep_clause

keep_clause ::= **keep** { cmp { **,** cmp } * }

The **keep** clause keeps in the working Dataset only the measures and attributes specified.

*Examples*

ds2 := **[** ds1**]** **{ keep** time, ref_area, partner, obs_value **}**

| ds1 | | | |
|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** |
| 2010 | EU25 | CA | 20 |
| 2010 | BG | CA | 2 |
| 2010 | RO | CA | 2 |

## filter_clause

filter_clause ::= **filter** boolean-expression | dpr

When a boolean expression is specified**, filter** filters out the data points of the working Dataset for which the boolean expression evaluates to false or null (i.e., only the data points for which the Boolean expression evaluates to true are maintained).
when a data point ruleset is specified**, filter** filters out the data points of the working Dataset for which at least one antecedent condition evaluates to true and its corresponding consequent condition evaluates to false or null (i.e., only the data points that satisfy the whole ruleset are maintained).
Note that **null** as a result of a boolean expression is always interpreted as "not satisfied".

*Examples*

1) Compute new measure obs_value_neg derived from obs_value, rename ds1.obs_status to keep it in the result. ds1.obs_value is not kept

ds_bop3 := **[outer** ds1, ds2 **] {**
**filter** ds2.obs_value <> 0 ,
obs_value = ds1.obs_value / ds2.obs_status ,
**rename** 'ds1.obs_status' to obs_status
**}**

| ds3 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE_NEG** | **OBS_STATUS** |
| 2010 | EU25 | CA | 2 | E |

2) Simple filter

ds2 := [ ds1 **] { filter** obs_value < 10 and time = "2010" **}**

| ds2 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | BG | CA | 2 | P |
| 2010 | RO | CA | 2 | P |

## rename_clause

rename_clause ::= **rename** cmp **to** cmp **{ ,** cmp **to** cmp }

**rename** allows renaming one or more components (Identifier , Measure or Attribute Component). VTL verifies that the resulting Dataset, after renaming all the specified components, has unique names of its components (otherwise an error is raised). Renaming an Identifier Componentimplies that the actual values of it are valid for the dimension type (usually the code list associated to the Identifier).

*Examples*

compute the measure obs_value_neg derived from obs_value
rename ds1.obs_status to keep it in the result
ds1.obs_value is not kept

ds_bop3 := **[ outer** ds1, ds2 **] {**
  obs_value_neg = **-**ds1.obs_value,
  **rename** 'ds1.obs_status' to obs_status        equivalent to obs_status = ds1.obs_status
  **}**

| ds3 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE_NEG** | **OBS_STATUS** |
| 2010 | EU25 | CA | -20 | E |
| 2010 | BG | CA | | P |
| 2010 | RO | CA | | P |

## unfold_clause

unfold_clause ::= **unfold** dim **,** msr **to** elem { **,** elem }

**unfold** creates the resulting Dataset in the following way: drops the Identifier Component *dim* and the measure *msr* from the resulting Dataset, partitions the input Dataset by grouping the values of the remaining Identifiers of the Dataset, transposes the data points of each group into a single data point of the resulting Dataset and adds new measures elements (all *elem* in the list). Then in the newly created data point **unfold** assigns to the value of each measure *elem* the value of *msr* existing in the input Dataset where dim = *elem* (if such a data point exists) or **null** otherwise.
The data points where "dim **not in** (*elem* , ...)" are removed from the resulting Dataset.
Note that the attributes created may have names that are not syntactically correct (they may start with a digit, contain special characters, etc.): those names must be quoted (included in single quote " **'** " ) in any expression, and it is not allowed to create a Dataset based on those data. It is also not allowed to return a Dataset as the final result of the join expression with names not complying with the VTL rules. Note that the names can be renamed using the **rename** operator.

*Examples*

Unfold and fold Identifier ref_area and measure obs_value
ds_unfold := **[** ds1 **] { unfold** ref_area, obs_value **to** (EU25, BG, RO ) **}**

| ds1 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 20 | E |
| 2010 | BG | CA | 2 | P |
| 2010 | RO | CA | 2 | P |

| ds_unfold | | | | |
|---|---|---|---|---|
| **TIME** | **PARTNER** | **EU25** | **BG** | **RO** |
| 2010 | CA | 20 | 2 | 2 |

### fold_clause

fold_clause ::= **fold** elem { **,** elem } **to** dim **,** msr

**fold** transposes a single data point of the input Dataset into several data points. It adds Identifier dim and measure msr to the resulting Dataset, inserts into the resulting Dataset a data point for each value A in the element list and assigns to the inserted data point dim = A and msr = value of measure A in the input Dataset. When measure A is null then **fold** does not create a data point for that measure.
Note that in general unfolding and folding are not exactly symmetric operations, i.e. in some cases the fold operation applied to the unfolded Dataset does not recreate exactly the original Dataset (before unfolding).

*Examples*
ds_fold := **[** ds_unfold **] {** **fold** (EU25, BG, RO ) **to** ref_area, obs_value **}**

| ds_fold | | | |
|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** |
| 2010 | EU25 | CA | 20 |
| 2010 | BG | CA | 2 |
| 2010 | RO | CA | 2 |

## Function Creation

*Semantics*
Creates a named function with given arguments, defined by a given expression.

*Syntax*
**create function** *function-name* **(** *arg-list* **)**
[ **returns** *return-type* ]
**as** *defining-expression*

*Parameters*
*function-name* : <IDENT>
*arg-list* : [ arg {, arg } ]
*arg* : arg-name [ **as** arg-type ] [ **:=** default-value ]
*arg-name* : <IDENT>
*arg-type* : type
*default-value* : literal
*defining-expression* : expression
*return-type* : type

*function-name* – the name under which the function is created
*arg-list* – the comma-separated list of formal arguments (can be empty)
*arg-name* – the name of an individual argument
*arg-type* – the optionally specified argument type
*default-value* – the optionally specified argument default value; it can be a scalar literal (number, string, Boolean, or date) or a function literal (an anonymous function)
*return-type* – the optionally specified function return type
*defining-expression* – the expression that defines the function

*Constraints*

- Each arg-name must be unique within the arg-list. For each arg-name, element arg-type can be omitted if the argument type can be inferred from the definition. If both arg-type and default-value are given, then default-value must be compatible with arg-type. Arguments that have default-value must come at the end of arg-list.
- If return-type is omitted, the statically inferred type of defining-expression is used as an implicit return-type. If return-type is given, the inferred type of defining-expression must be compatible with return-type.

*Returns*

Nothing

*Semantic specification*

The **create function** construct creates a named function with zero or more given arguments, defining expression, and the return type. The function can be called by name followed by the sequence of comma-separated call arguments in parentheses. Each call argument is an expression of type compatible with the corresponding *arg-type*, whose result is passed by value. The named function call syntax is:

*function-call* ::= function-name **(** call-arg-list **)**
*call-arg-list* ::= *{ call-arg {* **,** *call-arg } }*
*call-arg* ::= positional-arg | named-arg
*positional-arg* ::= arg-value
*named-arg* ::= *arg-name* **:=** arg-value
*arg-value* ::= expression

In *call-arg-list,* positional arguments and named arguments cannot be arbitrarily mixed: named arguments must come after all positional arguments (if any).
For *function-call* to be valid, the following properties are statically checked:
First, the *function-name* must refer to a function created with **create function.**
Second, all arguments to *function-name* that do not have *default-value* must be supplied. Positional arguments are supplied in the order in which they appear in the corresponding *arg-list*. The named arguments can be given in any order after the positional arguments, but cannot refer to arguments whose values are already given by an earlier *positional-arg* or *named-arg*.
For each *arg-name*, the type of the provided *arg-value* must be compatible with the corresponding *arg-type*.
Values for arguments with *default-value* must be specified using *named-arg*, nor *positional-arg.*
The result of a *function-call* to a *function-name* defined using **create function** is the value of *defining-expression* for the values of *arg-list* as supplied by the *call-arg-list*.

*Examples*

1)
**create function** compare_integer_descending(x **as integer**, y **as integer**)
**returns boolean**
**as** x > y
creates function *compare_integer_descending* which takes two integer arguments, *x* and *y*, and returns **true** if *x*>*y*, otherwise **false**. Call *compare_integer_descending*(1, 4) returns **false**, and call *compare_integer_descending*(8,0) returns **true**.

2)
**define function** has_solution(a, b, c)
**as** b*b-4*a*c>0
creates function *has_solution* takes three number arguments, *a, b* and *c*, and returns **true** only if the quadratic equation $ax^2$+*bx*+c=0 has at least one solution. The types for *a, b* and *c* are inferred as *number* because in the defining expression the left-hand side of > must be a number to be comparable with the right-hand side 0. Also, *return-type* is inferred as *Boolean*, because that is the result type for the comparison operator > on scalars.
Call *has_solution*(1,0,0) returns **true**, and *has_solution*(1,0,1) returns **false.** The latter is equivalent to *has_solution*(*a*:=1, *b*:=0, *c*:=1), which is also equivalent to *has_solution*(*c*:=1, *a*:=1, *b*:=0).

# VTL-ML - String operators and functions

## length

*Semantics*

The **length** operator returns the length of a character string.

*Syntax*

**length (** *ds* **)**

*Parameters*

ds : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as string-literal}+
{attribute <IDENT> as scalar-type}* | string]

*ds* – is a Dataset expression or a string

*Constraints*

If *ds* is a scalar then it must be a **string** type.
If *ds* is a Dataset then it must have at least a measure of type **string**.

*Returns*

If *ds* is a scalar then **length** returns a scalar **integer** representing the length of *ds*.
If *ds* is a Dataset and has N string measure components, then **length** returns a Dataset having the Identifier Components of *ds* and N numeric Measures with the same name as the string Measures of *ds* and containing the length of the corresponding measures.

*Examples*

On scalar

A := **length** ( "Hello, World!" )     A = 13

On Dataset

ds_r **:= length**(ds_1)

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | hello |
| 2 | B | null |

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 5 |
| 2 | B | null |

Note: the last value of M1 is null because the corresponding value of ds_1 is null.

## String concatenation ||

*Semantic*

The operator **||** concatenates two strings.

*Syntax*

*ds_1* || *ds_2*

*Parameters*

*ds_1*, *ds_2* : [ dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as string }+ {attribute <IDENT> as scalar }* | string]

*ds_1*,*ds_2* – is a Dataset expression or a string

*Constraints*

- If *ds_1* (*ds_2*) is a scalar then it must be a **string** data type.
- If *ds_1* (*ds_2*) is a Dataset then it has at least a measure of **string** type.
- If both *ds_1* and *ds_2* are Datasets then they must have at least one Identifier in common (with the same name and datatype).
- If both *ds_1* and *ds_2* are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

*Returns*

The operator returns:

If both *ds_1* and *ds_2* are scalar values then the || operator returns a scalar string value, the concatenation of *ds_1* and *ds_2*.

If either *ds_1* or *ds_2* is a Dataset then the || operator returns a Dataset having the following components:

- The superset of the Identifier Components of *ds_1* and *ds_2*
- If *ds_1* and *ds_2* have one or more string measures in common (i.e., with the same name) then the resulting Dataset has these common string measures, with the same name, containing the concatenation of the respective measures of *ds_1* and *ds_2*. Otherwise, if *ds_1* and *ds_2* do not have any measures in common and have only one measure then the resulting Dataset contains a measure named **CONDITION** that contains the concatenation of the single measures of *ds_1* and *ds_2*.

The resulting Dataset contains a data point for each pair of data points of *ds_1* and *ds_2* that have the same key (the same values of the Identifier Components).

*Examples*

On scalar

A := "Hello" || ", world! "    C = "Hello, world! "

On Dataset

ds_r := ds_1 || ds_2

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | "hello" |
| 2 | B | "hi" |

| ds_2 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | "world" |
| 2 | B | "there" |

| ds_r | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | "helloworld" |
| 2 | B | "hithere" |

## trim /rtrim/ltrim

*Semantics*

The **trim /rtrim/ltrim** operators eliminate trailing or/and leading whitespace from a string.

*Syntax*

[**trim** | **rtrim** | **ltrim**] **(** ds **)**

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as string }+
{attribute <IDENT> as scalar-type }* | string]

*ds* – is a Dataset expression or a string

*Constraints*

If *ds* is a scalar then it must be a **string** data type.
If *ds* is a Dataset then it must have at least a measure of **string** data type.

*Returns*

If *ds* is a scalar then operators returns a scalar string representing the input string without trailing or/and leading whitespace.
If *ds* is a Dataset and has N string measures then operators returns a Dataset having the Identifier Components of *ds* and N string measures with the same name as the string measures of *ds* where the values take the value of the input ones without whitespaces from left and right (trim), or alternatively without the left (ltrim) or right (rtrim) whitespaces.

*Semantic specification*

The operators trim whitespaces from left and right of it (trim), or alternatively only the left (ltrim) or right (rtrim) whitespaces.

*Examples*

example on scalar
If A = "  Hello, world!  ":
B := **trim(**A**)**     B = "Hello, world!"

example on Dataset
ds_1 := **trim(**ds**)**

| ds | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | " hello world " |
| 2 | B | "hi  " |
| 3 | C | " help! " |

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | "hello world" |
| 2 | B | "hi" |
| 3 | C | "help!" |

## upper/lower

*Semantics*
The **upper/lower** operators convert all characters of a string to upper / lower case.
*Syntax*
[**upper** | **lower**] **(** *ds* **)**

*Parameters*
*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as string }+
{attribute <IDENT> as scalar-type}*| string]

*ds* – is a Dataset expression or a string

*Constraints*
If *ds* is a scalar string then it must be a **string** data type.
If *ds* is a Dataset then it must have at least a measure of **string** data type.

*Returns*
If *ds* is a scalar then operators returns a scalar string that is the upper case or the lower case of the input one.
If *ds* is a Dataset and has N string measures, then the operators return a Dataset having the Identifier Components of *ds* and N string measures with the same name as the string measures of ds where the Measure Components assumes the upper case or lower case values of the respective input value of the Measure Components'.

*Examples*
On scalar
1) If A = "Hello, World!":
B := **upper(**A**)** B = "HELLO, WORLD!"
B := **lower(**A**)** B = "hello, world!"

On Dataset
2) ds_r := **upper(**ds_1**)**

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | "hello world" |
| 2 | B | "hi" |
| 3 | C | "help" |

| ds_r | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | "HELLO WORLD" |
| 2 | B | "HI" |
| 3 | C | "HELP" |

## substr

*Semantics*
The operator **substr** extracts a substring from a string

*Syntax*
**substr (** *ds*, {, *startPosition*} {, *length*} **)**

*Parameters*

*ds*: [dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as string }+
{attribute <IDENT> as scalar-type }*|string]
*startPostion* : integer
*length* : integer

- *ds* – is the input Dataset or the input string.
- *startPostion* – is the index of the character in the string from which the substring is performed.
- *length* – is the number of the characters in the string to be taken starting from *startPosition*.

### Constraints

- *startPostion* must be major or equal than 0 and minor than the whole length of the input string.
- *startPosition* plus length must be minor than the whole length of the input string, otherwise the length parameter is ignored.
- If ds is a scalar then it must be a **string** data type.
- If ds is a Dataset then it must have at least a measure of **string** data type.

### Returns

If *ds* is a scalar string then operators returns a substring of the input one starting from *startPosition* and extracting *length* characters.
If *ds* is a Dataset and has N string measures then operators returns a Dataset having the Identifier Components of *ds* and N string measures with the same name of the string measures of *ds* where the Measure Components assumes substring values of the respective input Measure Components's values, obtained starting from *startParameters* and taking length characters.

### Semantics

The substring of the input string is obtained stating from *startPosition* and extracting length characters, if *length* plus *startPosition* is greater than the whole length of the input string, then *length* parameter is ignored.

### Examples

On scalar
1) Assuming that A = "Hello, world!":

| | |
|---|---|
| B := **substr(**A, 2**)** | B = "lo, world!" |
| B := **substr(**A, 2, 5**)** | B = "lo, w" |
| B := **substr(**A, 0, 4**)** | B = "Hell" |

On Dataset
2) ds_r := **substr(**ds_1,7**)**

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | "hello world" |

| ds_r | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | "rld" |

3) ds_r := **substr(**ds_1,0,5**)**

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | "hello world" |

| ds_r | | |
|---|---|---|
| **K1** | **K2** | **M1** |

| | | |
|---|---|---|
| 1 | A | "hello" |

# instr

*Semantics*

The **instr** operator returns the position of a string in another one

*Syntax*

instr (*ds*, *strToSearch* {, *startPosition*} { , *occurrence*} )

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as string }+
{attribute <IDENT> as scalar-type }*|string]
*strToSearch* : string
*startPosition* : integer
*occurrence* : integer

- *ds* – is the input string or the input Dataset.
- *strToSearch* – is the string to search.
- *startPosition* – is the index of the character in the string from which start to search.
- *occurrence* – is the number of occurrences of the *strToSearch* from which start to search

*Constraints*

- If ds is a scalar then it must be a **string** data type.
- If ds is a Dataset then it must have at least a measure of **string** data type.

*Returns*

If *ds* is scalar then the operator returns the position of the first character of *strToSearch* in the string. The *startPosition* and *occurrence* are integers indicating the character of string and the number of occurrences from which start to search, respectively.
If *ds* is a Dataset and has N string measures then operators returns a Dataset having the IdentifierComponents of *ds* and N string measures with the same name of the string measures of *ds* where the Measure Components are integer representing the first character of *strToSearch* in the string. The *startPosition* and *occurrence* are integer indicating respectively the character of string and the number of occurrences from which start to search.
A negative value of startPosition counts backward from the end of string.

*Semantic specification*

If the string to search is not present in *str,* then the value returned is -1. If *startPosition* is omitted the start position is 1, if *occurrence* is omitted the value is 1.

*Examples*

On scalar
1) Assuming that A = "abcde":
B := **instr (**A, "c" **)** B = 2

On Dataset
2) ds_2 := **instr(**ds_1,"hello"**)**

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | "hello world" |
| 2 | A | "say hello" |
| 3 | A | "he" |
| 4 | A | "hi, hello!" |

| ds_2 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 0 |
| 2 | A | 4 |
| 3 | A | -1 |
| 4 | A | 4 |

## date_from_string

*Semantics*

The operator **date_from_string** converts a string into a date.

*Syntax*

**date_from_string(** *ds, format* **)**

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as string }+
{attribute <IDENT> as scalar-type }* |string]
*Format* : string

- *ds* – is the input string or the input Dataset
- *format* – is the format of the resulting date.

*Constraints*

- If ds is a scalar then it must be a **string** data type.
- If ds is a dataset then it must have at least a measure of **string** data type.
- format must respect one of these patterns:

| Format | Frequency | Example | Frequency |
|---|---|---|---|
| YYYY | | 2000 | Annual |
| YYYYSN | S | 2000S1 | Semestrial |
| YYYYQN | Q | 2000Q1 | Quarterly |
| YYYYMNN | M | 2000M01 | Monthly |
| YYYYDNNNN | D | 2000D0101 | Daily |
| YYYYA | A | 2000A | Annual |
| YYYYSN | S | 2000S1 | Semestrial |
| YYYY-QN | Q | 2000-Q1 | Quarterly |
| YYYY-NN | M | 2000-01 | Monthly |
| YYYY-NN-NN | D, M, Q or A | 2000-01-01 | Daily, Monthly, Quarterly or Annual |

*Returns*

If *ds* is a scalar, the operator returns its date representation, based on the chosen format.
If *ds* is a Dataset having N string Measure Components, the operator returns a Dataset having the same Identifier Components as *ds* and N Measure Components varying in type (from string-literal to date) assuming values of the date representations (on the base of the *format*) of the dates in the input Measure Components.

*Examples*

On scalar

1) If A = "2016-02"

B := **date_from_string** (A, YYYY-MM) B = 2016-02-01

2) If A = "2016-02"

B := **date_from_string** (A, YYYY-MM-DD) B = 2016-02-01

A date component has always years, months and days.

On Dataset

3) ds_2:= **date_from_string (**ds_1, “YYYY-MM”**)**

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | "2015-12" |
| 2 | B | "2015-06" |
| 3 | C | "2015-12" |
| 4 | E | "2015-06" |

| ds_2 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 2015-12-01 |
| 2 | B | 2015-06-01 |
| 3 | C | 2015-12-01 |
| 4 | E | 2015-06-01 |

## replace

*Semantics*

The **replace** operator replaces a substring with a given string.

*Syntax*

**replace(** *ds*, *str_old {, str_new }***)**

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> }+ {attribute <IDENT> as scalar-type }*|string]

*str_old, str_new* : string

- *ds* – is the input string or the input Dataset,
- *str_old* – is the string to be replaced,
- *str_new* – is the string to replace. If omitted then all occurrences of *str_old* are removed.

*Constraints*

- If ds is a scalar then it must be a **string** data type.
- If ds is a Dataset then it must have at least a measure of **string** data type.

*Returns*

- If ds is a scalar, the operator returns a string having the *ds* value obtained replacing *str_old* with *str_new*.
- If ds is a Dataset having N string Measure Components, returns a Dataset having the Identifier Component of ds and N string Measure Components obtained replacing *str_old* with *str_new*.

*Examples*

On scalar

1) If A = "Hello"

B := **replace** (A,"ello","i")    B = "Hi"

On Dataset

2) ds_2:= **replace** (ds_1,"ello","i")

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | “hello world” |
| 2 | A | “say hello” |
| 3 | A | “he” |
| 4 | A | “hello!” |

| ds_2 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | “hi world” |
| 2 | A | “say hi” |
| 3 | A | “he” |
| 4 | A | “hi!” |

# VTL-ML - Numeric operators and functions

## unary plus +

*Semantics*

The **+** operator leaves the sign unaltered.

*Syntax*

+ *ds*

*Parameters*

*ds* : [ dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*|number]

*ds* – is the input scalar number or the input Dataset.

*Constraints*

If *ds* is a scalar then it must be a **numeric** data type.
If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

*Returns*

If *ds* is a scalar then operator return the input number without altering its sign.
If *ds* is a Dataset and has N numeric measures then operator return a Dataset having the Identifier Components of *ds* and N numeric measures without alterations.

*Examples*

On scalar
1) A := +B
if B = 5, then A = 5

## unary minus -

*Semantics*

The **–** operator inverts the sign

*Syntax*

- *ds*

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*|number]

*ds* – is the input scalar number or the input Dataset.

*Constraints*

If *ds* is a scalar then it must be a **numeric** data type.
If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

*Returns*

If *ds* is a scalar then operator return the input number negated.
If *ds* is a Dataset and has N numeric measures then operator return a Dataset having the Identifier Components of *ds* and N numeric measures with the sign of the values in the numeric Measure Components inverted.

*Examples*

On scalar

1) A := **-**B

if B = 5, then A = -5

if B = -7, then A = 7

On Dataset

2) ds_2 := **-** ds_1

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 11 |
| 2 | B | -14 |
| 3 | C | 9 |

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | -11 |
| 2 | B | 14 |
| 3 | C | -9 |

3) ds_2 := - ds_1

| ds_1 | | | | |
|---|---|---|---|---|
| **K1** | **K2** | **M1** | **M2** | **M3** |
| 1 | A | 11 | 12 | “A” |
| 2 | B | -14 | 13 | “B” |
| 3 | C | 9 | 14 | “C” |

| ds_1 | | | | |
|---|---|---|---|---|
| **K1** | **K2** | **M1** | **M2** | **M3** |
| 1 | A | -11 | -12 | “A” |
| 2 | B | 14 | -13 | “B” |
| 3 | C | -9 | -14 | “C” |

## addition and subtraction + -

*Semantics*

The operator **+** or **–** compute the sum or subtraction

*Syntax*

*ds_1* [ + | - ] *ds_2*

*Parameters*

*ds_1, ds_2* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*|number]

- *ds_1*– is the first input scalar number or the first input Dataset.
- *ds_2* – is the second input scalar number or the second input Dataset.

### *Constraints*

- If *ds_1* (*ds_2*) is a scalar then it must be a **numeric** data type.
- If *ds_1* (*ds_2*) is a Dataset then it has at least a measure of **numeric** type.
- If both *ds_1* and *ds_2* are Datasets then they must have at least one Identifier Component in common (with the same name and data type).
- If both *ds_1* and *ds_2* are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

### *Returns*

If both *ds_1* and *ds_2* are scalar values then the operators return the algebraic sum or subtraction of *ds_1* and *ds_2*.
If either *ds_1* or *ds_2* is a Dataset then the operators return a Dataset having the following components:

- The superset of the Identifier Components of *ds_1* and *ds_2*
- If *ds_1* and *ds_2* have one or more numeric measures in common (i.e., with the same name) then the resulting Dataset has these common string measures, with the same name, containing the algebraic sum or subtraction of the respective measures of *ds_1* and *ds_2*. Otherwise, if *ds_1* and *ds_2* do not have any measures in common and have only one measure then the resulting Dataset contains only a measure named **CONDITION** that contains the algebraic sum or subtraction of the single measures of *ds_1* and *ds_2*.

The resulting Dataset contains a data point for each pair of data points of *ds_1* and *ds_2* that have the same values on the common Identifier Components).

### *Semantic specification*

See also the operator **listsum** than returns a data point for those data points that would be ignored.

### *Examples*

example on Dataset
1)
In this example, we calculate the total population of a set of countries given two Datasets: one of the male population, and another, of the female population. They contain one measure each. Thus, the result will contain a single measure with the results of the addition.

ds_3 := ds_1 **+** ds_2

| ds_1 | | |
|---|---|---|
| **TIME** | **GEO** | **POPULATION** |
| 2013 | Belgium | 5 |
| 2013 | Denmark | 2 |
| 2013 | France | 3 |
| 2013 | Spain | 4 |

| ds_2 | | | |
|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **POPULATION** |
| 2013 | Belgium | Total | 10 |
| 2013 | Greece | Total | 11 |
| 2013 | Belgium | Y15-24 | NULL |
| 2013 | Greece | Y15-24 | 2 |
| 2013 | Spain | Y15-24 | 6 |

| ds_3 | | | |
|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **POPULATION** |
| 2013 | Belgium | Total | 15 |
| 2013 | Belgium | Y15-24 | NULL |
| 2013 | Spain | Y15-24 | 10 |

Note that the Data Points of ds_1 and ds_2 that has a missing in the other Dataset are not shown in the resulting one.

2) ds_bop1 := ds_bop1 + 1

| ds_bop1 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 20 | D |
| 2010 | BG | CA | 2 | P |
| 2010 | RO | CA | 2 | P |

| ds_bop1 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 21 | D |
| 2010 | BG | CA | 3 | P |
| 2010 | RO | CA | 3 | P |

3) ds_plus := ds_bop1 + ds_bop2

| ds_bop1 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 20 | D |
| 2010 | BG | CA | 2 | P |
| 2010 | RO | CA | 2 | P |

| ds_bop2 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 10 | D |

| ds_plus | | | |
|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** |
| 2010 | EU25 | CA | 30 |

## multiplication and division * /

*Semantics*

The operator ***** or **/** multiply or divide two numbers.

*Syntax*

*ds_1* [ * | / ] *ds_2*

*Parameters*

*ds_1*, *ds_2* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+ {measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*|number]

- *ds_1*– is the first input scalar number or the first input Dataset.
- *ds_2* – is the second input scalar number or the second input Dataset.

*Constraints*

- If *ds_1* (*ds_2*) is a scalar then it must be a **numeric** data type.
- If *ds_1* (*ds_2*) is a Dataset then it has at least a measure of **numeric** type.
- If both *ds_1* and *ds_2* are Datasets then they must have at least a dimension in common (with the same name and data type).
- If both *ds_1* and *ds_2* are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

*Returns*

The operators return:
If both *ds_1* and *ds_2* are scalar values then the operators return the algebraic product or ratio of *ds_1* and *ds_2*.
If either *ds_1* or *ds_2* is a Dataset then the operators return a Dataset having the following components:

- The superset of the Identifier Components of *ds_1* and *ds_2*
- If *ds_1* and *ds_2* have one or more numeric measures in common (i.e., with the same name) then the resulting Dataset has these common numeric measures, with the same name, containing the algebraic sum or subtraction of the respective measures of *ds_1* and *ds_2*. Otherwise, if *ds_1* and *ds_2* do not have any measures in common and have only one measure then the resulting Dataset contains a measure named **CONDITION** that contains the algebraic product or ratio of the single measures of *ds_1* and *ds_2*.

The resulting Dataset contains a data point for each pair of data points of *ds_1* and *ds_2* that have the same key (the same values of the Identifier Components).
The algebraic product of the input numbers or the ratio between them.

*Semantic specification*

Division by zero results in a runtime exception.

*Examples*

On Dataset
DSr:=total_population[**rename** POPULATION **as** PERCENTAGE].PERCENTAGE * Overcrowding_rate_urbanization.PERCENTAGE

| total_population | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **POPULATION** | **UNEMPLOYMENT_RATE** |
| 2012 | Belgium | Total | 100 | 7.6 |
| 2012 | Greece | Total | 10 | 24.3 |
| 2012 | Spain | Total | 20 | 25 |
| 2012 | Belgium | Y15-24 | 30 | 3.6 |
| 2012 | Greece | Y15-24 | 5 | 18.3 |
| 2012 | Switzerland | Y15-24 | 2 | 20 |

| Overcrowding_rate_urbanization | | |
|---|---|---|
| **TIME** | **GEO** | **PERCENTAGE** |
| 2012 | Belgium | 0.01 |
| 2012 | Greece | 0.1 |
| 2012 | Spain | 0.2 |
| 2012 | Malta | 0.3 |
| 2012 | Finland | 0.4 |
| 2012 | France | 0.5 |

| DSr | | | |
|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **PERCENTAGE** |
| 2012 | Belgium | Total | 1 |
| 2012 | Greece | Total | 1 |
| 2012 | Spain | Total | 4 |
| 2012 | Greece | Y15-24 | 0.5 |

## round/ceil/floor

*Semantics*

The operators **round**/**ceil**/**floor** round a number.

*Syntax*

[**round(***ds*, *decimals***)** | **ceil(***ds***)** | **floor(***ds***)**]

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+ {attribute <IDENT> as scalar-type}*|number]

*decimals* : integer

- *ds* – is the input scalar number or the input Dataset.
- *decimals* – the decimal position to round to.

*Constraints*

If *ds* is a scalar then it must be a **numeric** data type.
If *ds* is a Dataset then it must have at least a measure of **numeric** data type.
*decimals* – must be an **integer** greater or equal than zero.

*Returns*

If *ds* is a scalar then the operators return the input number rounded using round, ceil or floor operator.
If *ds* is a Dataset and has N numeric measures then the operators return a Dataset having the Identifier Components of *ds* and N numeric measures rounded using round, ceil or floor operator.

*Semantic specification*

The operator **round** takes as input a number and a number of decimal digits and rounds the former number to the number of decimal digits specified by the latter.
The operator **floor** rounds to the largest previous integer, while ceil rounds to the smallest greater integer.

*Examples*

On scalar

1) If P = 3.14159

A := **round(**P, 2**)** A = 3.14
B := **round(**P, 4**)** B = 3.1416
C := **floor**(P) C = 3
D := **floor**(P) D = 4

On Dataset

| unemployment | | | | | |
|---|---|---|---|---|---|
| **AGE** | **TIME** | **GEO** | **SEX** | **YOUTH_UNEMPLOYMENT** | **UNEMPLOYMENT** |
| From 20 to 29 years | 2011 | Germany | Total | 7.5 | 5.9 |
| From 20 to 29 years | 2012 | Germany | Total | 7.1 | 5.5 |
| From 20 to 29 years | 2011 | Greece | Total | 33.7 | 17.7 |
| From 20 to 29 years | 2012 | Greece | Total | 42.5 | 24.3 |

2) ds_1 := **round(**unemployment,0**)**

| ds_1 | | | | | |
|---|---|---|---|---|---|
| **AGE** | **TIME** | **GEO** | **SEX** | **YOUTH_UNEMPLOYMENT** | **UNEMPLOYMENT** |
| From 20 to 29 years | 2011 | Germany | Total | 8 | 6 |
| From 20 to 29 years | 2012 | Germany | Total | 7 | 6 |
| From 20 to 29 years | 2011 | Greece | Total | 34 | 18 |
| From 20 to 29 years | 2012 | Greece | Total | 43 | 24 |

3) ds_1 := **round(**unemployment.YOUTH_UNEMPLOYMENT, 0**)**

| ds_1 | | | | | |
|---|---|---|---|---|---|
| **AGE** | **TIME** | **GEO** | **SEX** | **YOUTH_UNEMPLOYMENT** | **UNEMPLOYMENT** |
| From 20 to 29 years | 2011 | Germany | Total | 8 | 5.9 |
| From 20 to 29 years | 2012 | Germany | Total | 7 | 5.5 |
| From 20 to 29 years | 2011 | Greece | Total | 34 | 17.7 |
| From 20 to 29 years | 2012 | Greece | Total | 43 | 24.3 |

4) ds_1 := **ceil(**unemployment.YOUTH_UNEMPLOYMENT**)**

| ds_1 | | | | | |
|---|---|---|---|---|---|
| **AGE** | **TIME** | **GEO** | **SEX** | **YOUTH_UNEMPLOYMENT** | **UNEMPLOYMENT** |
| From 20 to 29 years | 2011 | Germany | Total | 8 | 6 |
| From 20 to 29 years | 2012 | Germany | Total | 7 | 6 |
| From 20 to 29 years | 2011 | Greece | Total | 34 | 18 |
| From 20 to 29 years | 2012 | Greece | Total | 43 | 25 |

## abs

*Semantics*

The operator **abs** calculates the absolute value of a number

*Syntax*

**abs(***ds***)**

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+
{attribute <IDENT> as scalar-type}* |number]

*ds* – is the input scalar number or the input Dataset.

*Constraints*

If *ds* is a scalar then it must be a **numeric** data type.
If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

*Returns*

If *ds* is a scalar then the operator returns the absolute value of the input number.
If *ds* is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier Components of *ds* and the N numeric measures containing the absolute values of the corresponding ones in *ds*.

*Examples*

On scalar

1) Let us assume A = -5:

B := **abs(**A**)** B = 5

C := **abs(**B**)** C = 5

On Dataset

2) DatasetB := **abs(**DatasetA**)**

| Dataset A | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 0.484183 |
| FR | Females | 2011 | -0.515817 |
| FR | Total | 2011 | -1.000000 |

| Dataset B | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 0.484183 |
| FR | Females | 2011 | 0.515817 |
| FR | Total | 2011 | 1.000000 |

3) ds_bop1 := **abs** ( ds_bop1 )

| ds_bop1 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 20 | D |
| 2010 | BG | CA | 2 | P |
| 2010 | RO | CA | -2 | P |

| ds_bop1 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |

| | | | | |
|---|---|---|---|---|
| 2010 | EU25 | CA | 20 | D |
| 2010 | BG | CA | 2 | P |
| 2010 | RO | CA | 2 | P |

# trunc

*Semantics*

The operator **trunc** truncates the decimal digits of a number.

*Syntax*

**trunc(***ds*, *decimals***)**

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+
{attribute <IDENT> as scalar-type}* |number]

*decimals* : integer

- *ds* – is the input scalar number or the input Dataset.
- *decimals* – the decimal position beyond which the decimal digits are discarded.

*Constraints*

If *ds* is a scalar then it must be a **numeric** data type.
If *ds* is a Dataset then it must have at least a measure of **numeric** data type.
*decimals* must be greater or equal than zero.

*Returns*

If *ds* is a scalar then the operator returns the input number with the decimal digits discarded beyond the number of digits specified by *decimals*.
If *ds* is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier Components of *ds* and the N numeric measures obtained by discarding the decimal digits after the *decimals* position.

*Examples*

On scalar

1) If P = 3.14159

A := **trunc(**P, 2**)** A = 3.14
B := **trunc(**P, 4**)** B = 3.141**5**

Differ from round:

B := **round(**P, 4**)** B = 3.141**6**

On Dataset

2) ds_1 := **trunc(**DatasetA, 2**)**

| DatasetA | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 0.484183 |
| FR | Females | 2011 | 0.515817 |
| FR | Total | 2011 | 1.000000 |

| ds_1 | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 0.48 |

| | | | |
|---|---|---|---|
| FR | Females | 2011 | 0.51 |
| FR | Total | 2011 | 1.00 |

## exp

*Semantics*

The **exp** operator calculates the exponential of a number

*Syntax*

**exp(***ds***)**

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}
{ measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}* |number]

*ds* – is the input scalar number or the input Dataset.

*Constraints*

If *ds* is a scalar then it must be a **numeric** data type.
If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

*Returns*

If *ds* is a scalar then the operator returns *e* (Napier's – or Euler's – constant) raised to *ds*.
If *ds* is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier components of *ds* and the N numeric measures obtained by elevating *e* (Nepero's number) to the value in the original Measure Component.

*Examples*

On scalar

1) If B = 5:
A := **exp(**B**)** A = 148.413
2) If B = -1:
A := **exp(**B**)** A = 0.368
3) If B = 0:
A := **exp(**B**)** A = 1.0

On Dataset

4) DatasetB := **exp(**DatasetA**)**

| Dataset A | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 5 |
| FR | Females | 2011 | 8 |
| FR | Total | 2011 | 2 |

| Dataset B | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 148.41 |
| FR | Females | 2011 | 2980.95 |
| FR | Total | 2011 | 7.389 |

## ln

*Semantics*

The operator **ln** calculates the natural logarithm of a number

*Syntax*

**ln(***ds***)**

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+
{attribute <IDENT> as scalar-type}* | number]

*ds* – is the input number.

*Constraints*

If *ds* is a scalar then it must be a **numeric** data type greater than zero.
If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

*Returns*

If *ds* is a scalar then the operator returns the natural logarithm (base *e*) of *ds*.
If *ds* is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier Components of *ds* and the N numeric measures obtained by calculating the natural logarithm (in base *e,* Nepero's number) of the value in the original Measure Component.

The logarithm of a zero or negative number results in a runtime exception.

*Examples*

On scalar

1) If B = 1:

A := **ln(**B**)** A = 0

2) If B = 148:

A := **ln(**B**)** A = 4.997

On Dataset

3) DatasetB := **ln(**DatasetA**)**

| Dataset A | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 148.41 |
| FR | Females | 2011 | 2980.95 |
| FR | Total | 2011 | 7.389 |

| Dataset B | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 5 |
| FR | Females | 2011 | 8 |
| FR | Total | 2011 | 2 |

# log

*Semantics*

The **log** operator calculates the logarithm of a number to a base b

*Syntax*

**log(***ds*, *base***)**

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+
{attribute <IDENT> as scalar-type}* |number]

*base* : integer

- *ds* – is the input scalar number (greater than zero) or the input Dataset.
- *base* – the base of the logarithm.

*Constraints*

- If *ds* is a scalar then it must be a **numeric** data type greater than zero.
- If *ds* is a Dataset then it must have at least a measure of **numeric** data type having values greater than zero.
- *base* must be greater than zero.

*Returns*

If ds is a scalar then the operator returns the *base* logarithm of ds.
If ds is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier Components of ds and the N numeric Measures are obtained by calculating the logarithm in *base* of the value in the original Measure Component.

The logarithm of a zero or negative number results in a runtime exception.

*Examples*

On scalar

1)If B = 1024:

A := **log**(B, 2)    A = 10

A := **log**(B, 10)    A = 3.01

On Dataset

2) DatasetB := **log(**2,DatasetA**)**

| Dataset A | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 1024 |
| FR | Females | 2011 | 64 |
| FR | Total | 2011 | 32 |

| Dataset B | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 10 |
| FR | Females | 2011 | 6 |
| FR | Total | 2011 | 5 |

# power

*Semantics*

The operator **power** calculates the power of a number raised to an exponent

*Syntax*
**power(***ds*, *exponent***)**

*Parameters*
*ds* : [ dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+ {attribute <IDENT> as scalar-type}* | number]
*exponent* : integer

- *ds* – is the input scalar number or the input Dataset.
- *exponent* – is the exponent of the power.

*Constraints*
- If *ds* is a scalar then it must be a **numeric** data type.
- If *ds* is a Dataset then it must have at least a measure of **numeric** data type.

*Returns*
If ds is a scalar then the operator returns ds raised to the *exponent* power.
If ds is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier. Components of ds and the N numeric measures are obtained by elevating the original Measure Component to the exponent-th power.

*Examples*
On scalar
1) If A = 2, B = 5:
C := **power(**B, A**)** C = 25

On Dataset
2) DatasetB := **power(**DatasetA,2**)**

| Dataset A | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 3 |
| FR | Females | 2011 | 4 |
| FR | Total | 2011 | 5 |

| Dataset B | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 9 |
| FR | Females | 2011 | 16 |
| FR | Total | 2011 | 25 |

## sqrt

*Semantics*
The operator **sqrt** calculates the square root of a number

*Syntax*
**sqrt(***ds***)**

*Parameters*
*ds :* [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+

{attribute <IDENT> as scalar-type}* |number]

*ds* – is the input scalar number or the input Dataset.

*Constraints*

- If *ds* is a scalar then it must be a **numeric** data type greater than zero.
- If *ds* is a Dataset then it must have at least a measure of **numeric** data type having values greater than zero.

*Returns*

If ds is a scalar then the operator returns the square root of *ds*.
If ds is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier. Components of ds and the N numeric measures are obtained by calculating the square root of the original Measure Component.

The square root of a negative number results in a runtime exception.

*Examples*

On scalar

1) If A = 25:

B := **sqrt(**A**)** B = 5

On Dataset

2) DatasetB := **sqrt(**DatasetA**)**

| Dataset A | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 16 |
| FR | Females | 2011 | 81 |
| FR | Total | 2011 | 64 |

| Dataset B | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 4 |
| FR | Females | 2011 | 9 |
| FR | Total | 2011 | 8 |

## nroot

*Semantics*

The **nroot** operator calculates the n-th root of a number

*Syntax*

**nroot(***ds, index***)**

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+
{attribute <IDENT> as scalar-type}* |number]
index : integer

- *ds* – is the input scalar number or the input Dataset.
- *index* – the index of the root.

*Constraints*

- If *ds* is a scalar then it must be a **numeric** data type greater than zero when index even (dynamic).

- If ds is a Dataset then it must have at least a measure of **numeric** data type having values greater than or equa to zero when index even (dynamic).

*Returns*

- If ds is a scalar then the operator returns the index-th root of ds.
- If ds is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier. Components of ds and the N numeric measures are obtained by calculating the index-th root of the original Measure Component.

*Semantic specification*

In case of even index and negative argument, it results in a runtime exception.

*Examples*

On scalar

1) If A = 2, B = 25:

C := **nroot**(B, A)          C = 5

On Dataset

2) DatasetB := **nroot**(DatasetA,3)

| Dataset A | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 8 |
| FR | Females | 2011 | 27 |
| FR | Total | 2011 | 64 |

| Dataset B | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 2 |
| FR | Females | 2011 | 3 |
| FR | Total | 2011 | 4 |

# mod

*Semantics*

The operator **mod** calculates the remainder of the division of a number by a denominator

*Syntax*

**mod(***ds*, *den***)**

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+
{attribute <IDENT> as scalar-type}* |number]

*den*: integer

- *ds* – is the input scalar number or the input Dataset.
- *den* – is the input denominator.

*Constraints*

- If *ds* is a scalar then it must be a **numeric** data type.
- If ds is a Dataset then it must have at least a measure of **numeric** data type.
- *den* must be greater than zero.

*Returns*

- If *ds* is a scalar then the operator returns the remainder of the division of *ds* by *den*.
- If *ds* is a Dataset and has N numeric measures then the operator returns a Dataset having the Identifier Components of *ds* and the N numeric measures are obtained by calculating the remainder of the division of the original Measure Component by *den*.

*Semantics*

The operator takes as input a numerator and a denominator and returns the remainder of the division of the numerator by the denominator.

*Examples*

On scalar

1) If A = 5, B = 2:

C := **mod(**A, B**)** C = 1

On Dataset

2) DatasetB := **mod(**DatasetA,3**)**

| Dataset A | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 7 |
| FR | Females | 2011 | 10 |
| FR | Total | 2011 | 12 |

| Dataset B | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 1 |
| FR | Females | 2011 | 1 |
| FR | Total | 2011 | 0 |

## listsum

*Semantics*

**listsum** returns the sum of the specified values and replaces the missing data points with a zero value

*Syntax*

**listsum (***ds* { **,** *ds* } * **)**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as number}+
{attribute <IDENT> as scalar-type}*

ds – is the input Dataset (s).

*Constraints*

The Dataset (s) must have at least a measure of **numeric** data type.

*Returns*

A Dataset denoting the sum of the values. If any expression evaluates to an empty data point then the 0 value is substituted for that expression. If all operands evaluate to empty data points then no data points are returned (i.e., the result is a Dataset containing no data points).

*Semantic specification*

The difference with the + operator is that **listsum** substitutes an empty data point with 0 (therefore returning a result) while the + operator returns an empty data point when one of the operands is an empty data point.

*Examples*

1) ds_sum := **listsum (** ds_bop1 , ds_bop2 **)**

| ds_bop1 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 20 | D |
| 2010 | BG | CA | 2 | P |
| 2010 | RO | CA | 2 | P |

| ds_bop2 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 10 | D |

| ds_sum | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 30 | D |
| 2010 | BG | CA | 2 | P |
| 2010 | RO | CA | 2 | P |

Compare with the "+" operator:

ds_plus := ds_bop1 **+** ds_bop2

| ds_plus | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 30 | D |

2) ds_sum := **listsum (** ds_bop1 , **-** ds_bop2 **)**

| ds_sum | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 10 | D |
| 2010 | BG | CA | 2 | P |
| 2010 | RO | CA | 2 | P |

# VTL-ML - Boolean operators and functions

## equal to =

*Semantic*

The operator **=** compares two values to evaluate if they are equal.

*Syntax*

*ds_1* **=** *ds_2*

*Parameters*

*ds_1*, *ds_2* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as <T>}+ | boolean]

*ds_1*,*ds_2* – is a Dataset expression or a boolean

*Constraints*

- If both *ds_1* and *ds_2* are Datasets then they must have at least a Identifier Component in common (with the same name and data type).
- If both *ds_1* and *ds_2* are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

*Returns*

If both *ds_1* and *ds_2* are scalar values then the **=** operator returns a scalar boolean value representing the result of the *equal to* validation.

If either *ds_1* or *ds_2* is a Dataset then the **=** operator returns a Dataset having the following components:

- The superset of the identifier components of *ds_1* and *ds_2*
- If *ds_1* and *ds_2* have one or more measures in common (i.e., with the same name) then the resulting Dataset has these common measures, with the same name concatenated with the suffix "**_CONDITION**", containing the results of the *equal to* validation of the respective measures of *ds_1* and *ds_2*. Otherwise, if *ds_1* and *ds_2* do not have any measures in common and have only one measure then the resulting Dataset contains a measure named **CONDITION** that contains the result of the *equal to* validation of the single measures of *ds_1* and *ds_2*.

The resulting Dataset contains a data point for each pair of data points of *ds_1* and *ds_2* that have the same key (the same values of the Identifier Components).

*Semantic specification*

If the two values are equal, the the result of the validation will be *true false* if they differ.

*Examples*

On scalar

1) If A = 5, B = 9, C = 5:

| | |
|---|---|
| D := A **=** B | D = false |
| D := A **=** C | D = true |

2) If A = "hello", B = "hi", C = "Hi":

| | |
|---|---|
| D := A **=** B | D = false |

On Dataset

3) DSr:=Overcrowding_rate_urbanization **=** 0.08

| overcrowding_rate_urbanization | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **VALUE** |
| 2012 | Belgium | Total | Total | NULL |
| 2012 | Greece | Total | Total | 0.286 |
| 2012 | Spain | Total | Total | 0.064 |

| | | | | |
|---|---|---|---|---|
| 2012 | Malta | Total | Total | 0.043 |
| 2012 | Finland | Total | Total | 0.08 |
| 2012 | Switzerland | Total | Total | 0.08 |

| DSr | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **CONDITION** |
| 2012 | Belgium | Total | Total | NULL |
| 2012 | Greece | Total | Total | false |
| 2012 | Spain | Total | Total | false |
| 2012 | Malta | Total | Total | false |
| 2012 | Finland | Total | Total | true |
| 2012 | Switzerland | Total | Total | true |

## not equal to <>

*Semantic*

The operator **<>** compares two values to evaluate if they are not equal.

*Syntax*

*ds_1* **<>** *ds_2*

*Parameters*

*ds_1*, *ds_2* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as <T>}+ | boolean]

*ds_1*,*ds_2* – is a Dataset expression or a boolean

*Constraints*

- If both *ds_1* and *ds_2* are Datasets then they must have at least a Identifier Component in common (with the same name and data type).
- If both *ds_1* and *ds_2* are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

*Returns*

If both *ds_1* and *ds_2* are scalar values then the operator returns a scalar Boolean value representing the result of the *not equal to* validation.
If either *ds_1* or *ds_2* is a Dataset then the operator returns a Dataset having the following components:

- The superset of the Identifier Components of *ds_1* and *ds_2*
- If *ds_1* and *ds_2* have one or more measures in common (i.e., with the same name) then the resulting Dataset has these common measures, with the same name concatenated with the suffix "**_CONDITION**", containing the results of the *not equal to* validation of the respective measures of *ds_1* and *ds_2*. Otherwise, if *ds_1* and *ds_2* do not have any measures in common and have only one measure then the resulting Dataset contains a measure named **CONDITION** that contains the result of the *not equal to* validation of the single measures of *ds_1* and *ds_2*.

The resulting Dataset contains a data point for each pair of data points of *ds_1* and *ds_2* that have the same key (the same values of the Identifier Components).

*Semantic specification*

If the values are not equals, the result of the validation will be *true*, *false* if they not differ.

*Examples*

On scalar

1) If A = 5, B = 9, C = 5:

D := A **<>** C D = false

2) If A = “hello”, B = “hi”, C = “Hi”:

D := A **<>** B D = true

On Dataset

3) compare_ds := y_unemployment_2012 **<>** y_unemployment_2011

| Y_unemployment_2012 | | | | |
|---|---|---|---|---|
| **GEO** | **SEX** | **UNIT** | **C_BIRTH** | **VALUE** |
| Germany | Total | Percentage | Total | 7.1 |
| Greece | Total | Percentage | Total | NULL |

| y_unemployment_2011 | | | | |
|---|---|---|---|---|
| **GEO** | **SEX** | **UNIT** | **C_BIRTH** | **VALUE** |
| Germany | Total | Percentage | Total | 7.5 |
| Greece | Total | Percentage | Total | 3 |

| compare_ds | | | | |
|---|---|---|---|---|
| **GEO** | **SEX** | **UNIT** | **C_BIRTH** | **VALUE_CONDITION** |
| Germany | Total | Percentage | Total | true |
| Greece | Total | Percentage | Total | NULL |

If VALUE for Greece in the second operand had also been NULL, then the result would still be NULL for Greece.

## greater than > >=

*Semantic*

The operator **> >=** compares two values to evaluate if one is greater (or equal) to the other.

*Syntax*

*ds_1* [ **>**| **>=**] *ds_2*

*Parameters*

*ds_1*, *ds_2* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as <T>}+ | boolean]

*ds_1*,*ds_2* – is a Dataset expression or a Boolean

*Constraints*

- If both *ds_1* and *ds_2* are Datasets then they must have at least one Identifier Componentin common (with the same name and data type).
- If both *ds_1* and *ds_2* are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

*Returns*

If both *ds_1* and *ds_2* are scalar values then the operator returns a scalar boolean value representing the result of the comparison *greater* (or *equal*) validation.
If either *ds_1* or *ds_2* is a Dataset then the operator returns a Dataset having the following components:

- The superset of the Identifier Components of *ds_1* and *ds_2*
- If *ds_1* and *ds_2* have one or more measures in common (i.e., with the same name) then the resulting Dataset has these common measures, with the same name concatenated with the suffix “**_CONDITION**”, containing the results of the comparison greater (or equal) validation of the respective measures of *ds_1* and *ds_2*. Otherwise, if *ds_1* and *ds_2* do not have any measures in common and have only one measure then the

resulting Dataset contains a measure named **CONDITION** that contains the result of the comparison greater (or equal) validation of the single measures of *ds_1* and *ds_2*.

The resulting Dataset contains a data point for each pair of data points of *ds_1* and *ds_2* that have the same key (the same values of the Identifier Components).

*Semantic specification*

If the value on the left side is greater (or equal) than the value on the right side, the result of the validation will be *true* if, *false* if not or either of them is NULL.

*Examples*

On scalar

1) If A = 5, B = 9, C = 5:

D := A **>** B D = false

D := A **>=** C D = true

2) If A = "hello", B = "hi", C = "Hi":

D := A **>** B D = false

On Dataset

3) compare_ds := foreign_languages_known > 20

| foreign_languages_known | | | | | |
|---|---|---|---|---|---|
| **N_LANG** | **GEO** | **TIME** | **AGE** | **UNIT** | **VALUE** |
| 2 | Germany | 2011 | Total | Percentage | NULL |
| 2 | Greece | 2011 | Total | Percentage | 12.2 |
| 2 | Finland | 2011 | Total | Percentage | 29.5 |

| compare_ds | | | | | |
|---|---|---|---|---|---|
| **N_LANG** | **GEO** | **TIME** | **AGE** | **UNIT** | **CONDITION** |
| 2 | Germany | 2011 | Total | Percentage | NULL |
| 2 | Greece | 2011 | Total | Percentage | false |
| 2 | Finland | 2011 | Total | Percentage | true |

4) compare_ds := y_unemployment_2012 **>** y_unemployment_2011

| y_unemployment_2012 | | | | |
|---|---|---|---|---|
| **GEO** | **SEX** | **UNIT** | **C_BIRTH** | **VALUE** |
| Germany | Total | Percentage | Total | 7.1 |
| Greece | Total | Percentage | Total | 42.5 |

| y_unemployment_2011 | | | | |
|---|---|---|---|---|
| **GEO** | **SEX** | **UNIT** | **C_BIRTH** | **VALUE** |
| Germany | Total | Percentage | Total | 7.5 |
| Greece | Total | Percentage | Total | 33.7 |

| compare_ds | | | | |
|---|---|---|---|---|
| **GEO** | **SEX** | **UNIT** | **C_BIRTH** | **VALUE_CONDITION** |
| Germany | Total | Percentage | Total | false |
| Greece | Total | Percentage | Total | true |

If the VALUE column for Germany in the y_unemployment_2012 Dataset had a NULL value the result would be:

| compare_ds | | | | |
|---|---|---|---|---|
| **GEO** | **SEX** | **UNIT** | **C_BIRTH** | **VALUE_ CONDITION** |
| Germany | Total | Percentage | Total | NULL |
| Greece | Total | Percentage | Total | true |

## less than < <=

*Semantic*

The operator **< <=** compares two values to evaluate if one is less (or equal) to the other.

*Syntax*

*ds_1* [ **<**| **<=**] *ds_2*

*Parameters*

*ds_1*, *ds_2* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as <T>}+ | boolean]

*ds_1*,*ds_2* – is a Dataset expression or a boolean

*Constraints*

- If both *ds_1* and *ds_2* are Datasets then they must have at least one Identifier Component in common (with the same name and data type).
- If both *ds_1* and *ds_2* are Datasets then either they have one or more measures in common, or at least one of them has only a measure.

*Returns*

If both *ds_1* and *ds_2* are scalar values then the operator returns a scalar Boolean value representing the results of the comparison *less* (or *equal*) than validation.
If either *ds_1* or *ds_2* is a Dataset then the operator returns a Dataset having the following components:

- The superset of the Identifier Components of *ds_1* and *ds_2*
- If *ds_1* and *ds_2* have one or more measures in common (i.e., with the same name) then the resulting Dataset has these common measures, with the same name concatenated with the suffix "**_CONDITION**", containing the results of the comparison less (or equal) than validation of the respective measures of *ds_1* and *ds_2*. Otherwise, if *ds_1* and *ds_2* do not have any measures in common and have only one measure then the resulting Dataset contains a measure named **CONDITION** that contains the results of the comparison less (or equal) than validation of the single measures of *ds_1* and *ds_2*.

The resulting Dataset contains a data point for each pair of data points of *ds_1* and *ds_2* that have the same key (the same values of the Identifier Components).

*Semantic specification*

*f* the value on the left side is less (or equal) than the value on the right side the result of the validation will be *true*, *false* if not or if either of them is NULL.

*Examples*

On scalar
1) If A = 5, B = 9, C = 5:
D := A **<** B D = true
D := A **<=** C D = true
2) If A = "hello", B = "hi", C = "Hi":
D := C **<** B D = false

On Dataset
3) compare_ds := total_population < 15000000

| total_population | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **VALUE** |

| | | | | |
|---|---|---|---|---|
| 2012 | Belgium | Total | Total | 11094850 |
| 2012 | Greece | Total | Total | 11123034 |
| 2012 | Spain | Total | Total | 46818219 |
| 2012 | Malta | Total | Total | NULL |
| 2012 | Finland | Total | Total | 5401267 |
| 2012 | Switzerland | Total | Total | 7954662 |

| compare_ds | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **CONDITION** |
| 2012 | Belgium | Total | Total | true |
| 2012 | Greece | Total | Total | true |
| 2012 | Spain | Total | Total | false |
| 2012 | Malta | Total | Total | NULL |
| 2012 | Finland | Total | Total | true |
| 2012 | Switzerland | Total | Total | true |

# in, not in

*Semantic*

The operator **in**, **not in** verifies if a value belongs to a set of values of a set or a list

*Syntax*

*ds* {**not**} **in** [*list* | *inlineList* ]

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as <T>}+|constant<T>]
*list* : list-ref
inlineList : list( {constant<T>}+ )

- *ds* – is a Dataset expression or a scalar
- *list* – is a reference to a valid List.
- *inlineList* – is an in-line specification of a List. The elements of the List are constants.

*Constraints*

- if *ds* is a scalar the elements of the List must be of the same type and If set is specified, then it must be a reference to a mono-dimensional Set.
- if *ds* is a Dataset, all the Measure Components of ds must have the same type T (which is also the type of the Set or List),

*Returns*

If *ds* is a scalar then **in, not in** returns a Boolean value representing the presence of the constant in the List.
If *ds* is a Dataset and has N measures then **in, not in** returns a Dataset having the identifier components of *ds* and N Boolean Measure Components having the same name concatenated with the suffix "**_CONDITION**" that states if the values of *ds* are (not) in the *list*.

*Examples*

On Dataset
ds_1 := total_population **in** (11094850, 46818219, 222, 111)

| total_population | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 11094850 |

| | | | | |
|---|---|---|---|---|
| 2012 | Greece | Total | Total | 11123034 |
| 2012 | Spain | Total | Total | 46818219 |
| 2012 | Malta | Total | Total | 417546 |
| 2012 | Finland | Total | Total | 5401267 |
| 2012 | NULL | Total | Total | 7954662 |

| ds_1 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION_CONDITION** |
| 2012 | Belgium | Total | Total | true |
| 2012 | Greece | Total | Total | false |
| 2012 | Spain | Total | Total | true |
| 2012 | Malta | Total | Total | false |
| 2012 | Finland | Total | Total | false |
| 2012 | NULL | Total | Total | false |

# between

*Semantic*

The operator **between** verifies if a value belongs to an interval of values

*Syntax*

*ds_1* **between** *ds_2* **and** *ds_3*

*Parameters*

*ds_1, ds_2, ds_3* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as <T>}+ |constant<T>]

- *ds_1* – is the Dataset or the scalar to validate.
- *ds_2* – is the lowerbound of a value's range.
- *ds_3* – is the upperbound of a value's range.

*Constraints*

- If *ds_1* is a scalar then the defined constants must be all of the same type.
- If *ds_1* is a Dataset then:
  - At least one Dataset must be defined.
  - If two (or three) Datasets are defined, for every pair of Datasets, it must hold that either they have the same Identifier Components or the ones of the former is a subset of the ones of the latter (static).
  - If two (or three) Datasets are defined, they must have the same Measure Components, in name and number (as explained in the syntax) (static).
  - If at least one Constant is defined, the Datasets must have a single Measure Component of type <T> (static).

*Returns*

If *ds_1* is a scalar then **between** returns a Boolean value representing if *c_1* is greater or equal than *c_2* and less or equal than *c_3.*

If *ds* is a Dataset and has N measures then operator returns a Dataset having the Identifier Components of *ds* and N Boolean measures having the same name concatenated with the suffix "**_CONDITION**" containing the result of the range comparison.

*Examples*

On Dataset

1) comparison_ds := unemployment_rate **between** 7.5 **and** 8.0

| unemployment_rate | | |
|---|---|---|
| **TIME** | **GEO** | **UNEMPLOYMENT_RATE** |
| 2013 | Finland | 8.2 |
| 2012 | Finland | 7.7 |
| 2011 | Finland | 7.8 |
| 2010 | Finland | 8.4 |
| 2009 | Finland | NULL |

| comparison_ds | | |
|---|---|---|
| **TIME** | **GEO** | **UNEMPLOYMENT_RATE** |
| 2013 | Finland | false |
| 2012 | Finland | true |
| 2011 | Finland | true |
| 2010 | Finland | false |
| 2009 | Finland | NULL |

2) comparison_ds := overcrowding_rate_urbanization_2011 **between** Overcrowding_rate_urbanization_2010 **and** Overcrowding_rate_urbanization_2012

| overcrowding_rate_urbanization_2011 | |
|---|---|
| **GEO** | **VALUE** |
| Belgium | NULL |
| Greece | 0.276 |
| Finland | 0.093 |
| Switzerland | 0.08 |
| United Kingdom | 0.089 |
| France | 0.125 |

| overcrowding_rate_urbanization_2010 | | | |
|---|---|---|---|
| **GEO** | **AGE** | **SEX** | **VALUE** |
| Belgium | Total | Total | 0.06 |
| Greece | Total | Total | 0.281 |
| Spain | Total | Total | 0.06 |
| Malta | Total | Total | 0.041 |
| Switzerland | Total | Total | NULL |

| overcrowding_rate_urbanization_2012 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **VALUE** |
| 2012 | Belgium | Total | Total | 0.023 |
| 2012 | Greece | Total | Total | 0.286 |
| 2012 | Spain | Total | Total | 0.064 |
| 2012 | Malta | Total | Total | 0.043 |
| 2012 | Finland | Total | Total | 0.08 |

| | | | | |
|---|---|---|---|---|
| 2012 | Switzerland | Total | Total | 0.08 |

| comparison_ds | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **VALUE_CONDITION** |
| 2012 | Belgium | Total | Total | NULL |
| 2012 | Greece | Total | Total | false |
| 2012 | Switzerland | Total | Total | NULL |

# isnull

*Semantics*

The **isnull** operator, compares the values with the NULL.

*Syntax*

**isnull(***ds***)**

*Parameters*

*ds* : [ dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type }*
{attribute <IDENT> as scalar-type}*|constant]

*ds* – is a Dataset or a scalar value.

*Constraints*

None

*Returns*

If *ds* is a scalar then **isnull** returns a boolean value representing if the value is (not) NULL.
If *ds* is a Dataset and has N measures then **isnull** returns a Dataset having the Identifier Components of *ds* and N numeric measures with the same name concatenated with the suffix "**_CONDITION**" but assuming a boolean value if the value is (not) NULL.

*Examples*

On scalar

1) If C is null:

A := **isnull(**C**)** A = true

2) If C is not null:

A := **isnull(**C**)** A = false

On Dataset

3) ds_1 := **isnull(**population**)**

| population | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 11094850 |
| 2012 | Greece | Total | Total | 11123034 |
| 2012 | Spain | Total | Total | NULL |
| 2012 | Malta | Total | Total | 417546 |
| 2012 | Finland | Total | Total | 5401267 |
| 2012 | NULL | Total | Total | NULL |

| ds_1 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **CONDITION** |
| 2012 | Belgium | Total | Total | false |
| 2012 | Greece | Total | Total | false |
| 2012 | Spain | Total | Total | true |
| 2012 | Malta | Total | Total | false |
| 2012 | Finland | Total | Total | false |
| 2012 | NULL | Total | Total | true |

# exists_in, not_exists_in/in_all

*Semantics*

The **exists_in, not_exists_in/in_all** operators match the existence or not of data points of a Dataset in another Dataset.

*Syntax*

*ds_1* [**exists_in**|**exists_in_all**|**not_exists_in**|**not_exists_in_all** ] *ds_2*

Parameters

*ds_1, ds_2* : dataset {identifier <IDENT> as <T>}+ {measure <IDENT> as <T> }*
{attribute <IDENT> as scalar-type}*

*ds_1, ds_2* – are the input Datasets.

*Constraints*

*ds_1* and *ds_2* must have at least one Identifier Component in common (with the same name and data type).

*Returns*

A Dataset with all Identifier Components of the two Datasets and one boolean Measure Component named **CONDITION**. The Measure Component value in each Data Points in the output indicates whether a Data Point with matching key (not) exists in the second argument for the corresponding Data Point of the first argument.

*Semantic specification*

If **all** versions are used, both the *true* and the *false* Data Points are kept in the result. Otherwise, only the *true* Data Points are kept.

*Examples*

1) ds_check := population **exists_in_all** urbanization_rate

| population | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 11094850 |
| 2012 | Greece | Total | Total | 11123034 |
| 2012 | Spain | Total | Total | 46818219 |
| 2012 | Malta | Total | Total | 417546 |
| 2012 | Finland | Total | Total | 5401267 |
| 2012 | Switzerland | Total | Total | 7954662 |

| urbanization_rate | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **RATE** |

| | | | | |
|---|---|---|---|---|
| 2012 | Belgium | Total | Total | 0.023 |
| 2012 | Greece | Total | Total | 0.286 |
| 2012 | Spain | Total | Total | 0.064 |
| 2012 | Malta | Total | Total | 0.043 |
| 2012 | Finland | Total | Total | NULL |
| 2012 | Switzerland | Total | Total | 0.08 |

| ds_check | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **CONDITION** |
| 2012 | Belgium | Total | Total | true |
| 2012 | Greece | Total | Total | false |
| 2012 | Spain | Total | Total | true |
| 2012 | Malta | Total | Total | false |
| 2012 | Finland | Total | Total | false |
| 2012 | Switzerland | Total | Total | true |

2) R := C1 **exists_in** C2

| C1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 100 |
| 2 | B | 200 |
| 3 | C | 700 |
| 4 | A | 550 |
| 5 | D | 120 |

| C2 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 100 |
| 2 | B | 200 |
| 5 | D | 700 |

| R | | |
|---|---|---|
| **K1** | **K2** | **CONDITION** |
| 1 | A | True |
| 2 | B | True |
| 5 | D | True |

3) R := C1 **exists_in_all** C2

| C1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 100 |

| | | |
|---|---|---|
| 2 | B | 200 |
| 3 | C | 700 |
| 4 | A | 550 |
| 5 | D | 120 |

| C2 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 100 |
| 2 | B | 200 |
| 5 | D | 700 |

| R | | |
|---|---|---|
| **K1** | **K2** | **CONDITION** |
| 1 | A | True |
| 2 | B | True |
| 3 | C | False |
| 4 | A | False |
| 5 | D | True |

4) R := C1 does **not_exist_in** C2

| C1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 100 |
| 2 | B | 200 |
| 3 | C | 700 |
| 4 | A | 550 |
| 5 | D | 120 |

| C2 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 100 |
| 2 | B | 200 |
| 5 | D | 700 |

| R | | |
|---|---|---|
| **K1** | **K2** | **CONDITION** |
| 3 | C | True |
| 4 | A | True |

5) R := C1 **not_exists_in_all** C2

| C1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 100 |
| 2 | B | 200 |
| 3 | C | 700 |
| 4 | A | 550 |
| 5 | D | 120 |

| C2 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 100 |
| 2 | B | 200 |
| 5 | D | 700 |

| R | | |
|---|---|---|
| **K1** | **K2** | **CONDITION** |
| 1 | A | False |
| 2 | B | False |
| 3 | C | True |
| 4 | A | True |
| 5 | D | False |

6) R := C1 **not_exists_in_all** C2

| C2 | | | |
|---|---|---|---|
| **K1** | **K2** | **K3** | **M1** |
| 1 | A | X | 100 |
| 2 | B | Y | 200 |
| 5 | D | Z | 700 |
| 5 | D | K | 1500 |

| C1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 100 |
| 2 | B | 200 |
| 3 | C | 700 |
| 4 | A | 550 |
| 5 | D | 120 |

| R | | |
|---|---|---|
| **K1** | **K2** | **CONDITION** |

| | | |
|---|---|---|
| 1 | A | False |
| 2 | B | False |
| 3 | C | True |
| 4 | A | True |
| 5 | D | False |

# match_characters

*Semantics*

The **match_character** operator checks whether a value respects a given pattern

*Syntax*

**Match_characters (** *ds*, *pattern* {, **all**}**)**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as string }*
{attribute <IDENT> as scalar-type}*
*pattern :* regexp
*regexp* : string

- *ds* – is the input Dataset.
- *pattern* – is a regular expression that defines a string pattern.
- *regexp* – is a regular expression.

*Constraints*

- *ds* must have only one **string** Measure Components (static).
- *pattern* must be a regular expression according to POSIX extended standard ([http:pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap09.html](http:pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap09.html)) (static).

*Returns*

A Dataset having the same Identifier and Attribute Components and a Boolean Measure Component for each string Measure Component in *ds* with the same name concatenated with the suffix "**_CONDITION**", containing the value resulting from the matching between the values in *ds* and the specified *pattern.*

*Semantic specification*

The Data Points of *ds* are copied into the output Dataset; the Boolean Measure Component will have *true* if the respective in *ds* matches with the *pattern*, *false* otherwise.
The **all** flag allows to specify that both *true* and *false* Data Points have to be kept in the output. If it is not present, only *true* Data Points are kept.

*Examples*

ds_r := **match_characters(**population.TIME,"[123456789,]",**all)**

| population | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 11094850 |
| 2012A | Greece | Total | Total | 11123034 |
| 2012 | Spain | Total | Total | 46818219 |
| 2012 | Malta | Total | Total | 417546 |
| 2012 | Finland | Total | Total | 5401267 |
| 2012C | Switzerland | Total | Total | 7954662 |

| population | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION_CONDITION** |
| 2012 | Belgium | Total | Total | true |
| 2012A | Greece | Total | Total | false |
| 2012 | Spain | Total | Total | true |
| 2012 | Malta | Total | Total | true |
| 2012 | Finland | Total | Total | true |
| 2012C | Switzerland | Total | Total | false |

# all

*Semantics*

The **all** operator verifies that all values in the Dataset are true

*Syntax*

**all (***ds***)**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as boolean }+
{attribute <IDENT> as scalar-type }*

*ds* – is a Dataset

*Constraints*

*ds* must have at least a measure of type **boolean**.

*Returns*

A Dataset with only one Boolean measure, called **CONDITION**, equal to *true* if, for all data points of the input Dataset, the Boolean measures are equal to *true*, *false* otherwise.

*Examples*

1) ds_2 := **all**( ds_1.VALUE>100 )

| ds_1 | | |
|---|---|---|
| **GENDER** | **TIME** | **VALUE** |
| M | 2000 | 200 |
| F | 2000 | 50 |
| M | 2001 | 150 |
| F | 2001 | 120 |

| ds_2 |
|---|
| **CONDITION** |
| FALSE |

2) ds_2 := **all**( ds_1[filter time= "2001"].VALUE>100 )

| ds_1 | | |
|---|---|---|
| **GENDER** | **TIME** | **VALUE** |
| M | 2000 | 200 |
| F | 2000 | 50 |
| M | 2001 | 150 |
| F | 2001 | 120 |

| ds_2 |
|---|
| **CONDITION** |
| TRUE |

3) ds_2 := **all**(ds_1.VALUE_2000>100 AND ds_1.VALUE_2001>100 )

| ds_1 | | |
|---|---|---|
| **GENDER** | **VALUE_2000** | **VALUE_2001** |
| M | 200 | 150 |
| F | 50 | 120 |

| **ds_2** |
|---|
| **CONDITION** |
| FALSE |

## any

*Semantics*

The **any** operator verifies that at least one value in the Dataset is true

*Syntax*

**any (***ds***)**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as boolean}+
{attribute <IDENT> as scalar-type }*

*ds* – is a Dataset

*Constraints*

*ds* must have at least one measure of type **Boolean**.

*Returns*

A Dataset with only one Boolean measure,called **CONDITION**, equal to *true* if, for at least one data point of the input Dataset, the Boolean measures are equal to *true*, *false* otherwise.

*Examples*

1) ds_2 := **any**( ds_1.VALUE>100 )

| ds_1 | | |
|---|---|---|
| **GENDER** | **TIME** | **VALUE** |
| M | 2000 | 200 |
| F | 2000 | 50 |
| M | 2001 | 90 |
| F | 2001 | 120 |

| ds_2 |
|---|
| **CONDITION** |
| TRUE |

2) ds_2 := **any** (ds_1.VALUE_2000>100 AND ds_1. VALUE_2001>100 )

| ds_1 | | |
|---|---|---|
| **GENDER** | **VALUE_2000** | **VALUE_2001** |
| M | 200 | 90 |
| F | 50 | 120 |

| ds_2 |
|---|
| **CONDITION** |
| FALSE |

3) ds_2 := **any** (ds_1.VALUE_2000>100 AND ds_1.VALUE_2001>100)

| ds_1 | | |
|---|---|---|
| **GENDER** | **VALUE_2000** | **VALUE_2001** |
| M | 200 | 90 |
| F | 50 | 120 |

| ds_2 |
|---|
| **CONDITION** |
| TRUE |

# unique

*Semantics*

The **unique** operator verifies the presence of one single Data Point having true as the value for the Measure component.

*Syntax*

**unique (***ds***)**

*Parameters*

ds : dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as boolean }+
{attribute <IDENT> as scalar-type }*

*ds* – is a Dataset

*Constraints*

*ds* must have at least a measure of type **boolean.**

*Returns*

A Dataset with only one boolean measure,called **CONDITION**, equal to *true* if, for only one data point of the input Dataset, the boolean measures are is equal to *true*, *false* otherwise.

*Examples*

1) ds_2 := **unique** ( ds_1.VALUE>100 )

| ds_1 | | |
|---|---|---|
| **GENDER** | **TIME** | **VALUE** |
| M | 2000 | 200 |
| F | 2000 | 150 |
| M | 2001 | 90 |
| F | 2001 | 120 |

| ds_2 |
|---|
| **CONDITION** |
| FALSE |

2) ds_2 := **unique** (ds_1.VALUE_2000>100 AND ds_1.VALUE_2001>100 )

| ds_1 | | |
|---|---|---|
| **GENDER** | **VALUE_2000** | **VALUE_2001** |
| M | 200 | 90 |
| F | 150 | 120 |

| ds_2 |
|---|
| **CONDITION** |
| TRUE |

3) ds_2 := **unique** (ds_1.VALUE_2000>100 AND ds_1.VALUE_2001>100 )

| ds_1 | | |
|---|---|---|
| **GENDER** | **VALUE_2000** | **VALUE_2001** |
| M | 200 | 90 |
| F | 150 | 120 |

| ds_2 |
|---|
| **CONDITION** |
| FALSE |

# func_dep

*Semantics*

The **func_dep** operator checks the functional dependency between components of a Dataset.

*Syntax*

**func_dep (** *ds* **,** *listCompFrom* **,** *listCompTo* **)**

*listCompFrom* : (comp { **,** comp } *)
*listCompTo* : (comp { **,** comp } *)

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as scalar-type }+ {attribute <IDENT> as scalar-type }*

*listCompFrom* –the components that form the left side of the functional dependency
*listCompTo* – the components that form the right side of the functional dependency

*Constraints*

None

*Returns*

A Dataset having the only measure **CONDITION**, assuming value *true* if the functional dependency between the left and the right side is respected.

*Semantic specification*

The *func_dep* operator verifies the existence of a functional dependency from the components in *listCompFrom* to the the components in *listCompTo* ( *listCompFrom* → *listCompTo* ), that is, each combination of values of the components *listCompFrom* corresponds to one combination of values of the components *listCompTo*.

*Examples*

1) ds_2 := **func_dep (** ds_1, (FISCAL_CODE), (NAME)**)**

| ds_1 | | | |
|---|---|---|---|
| **FISCAL_CODE** | **NAME** | **DATE_OF_BIRTH** | **PLACE_OF_BIRTH** |
| FC1 | John Smith | 10/09/1968 | London |
| FC2 | Helen Brown | 18/10/1976 | London |

| | | | |
|---|---|---|---|
| FC3 | Steve McGill | 21/08/1966 | Dublin |
| FC4 | Helen Brown | 26/02/2001 | Dublin |

| ds_2 |
|---|
| **CONDITION** |
| TRUE |

2) ds_3:= **func_dep (** ds_1, (FISCAL_CODE ), (NAME, DATE_OF_BIRTH,PLACE_OF_BIRTH)**)**

| ds_3 |
|---|
| **CONDITION** |
| TRUE |

3) ds_2 := **func_dep (** NAME), (FISCAL_CODE) **)**

| ds_2 |
|---|
| **CONDITION** |
| FALSE |

## and

*Semantics*
The **and** operator calculates the logical AND

*Syntax*
*ds_1* **and** *ds_2*

*Parameters*
*ds_1, ds_2* : [dataset {identifier <IDENT> as scalar-type }+ {measure <IDENT> as boolean}+ {attribute <IDENT> as scalar-type}*|boolean]

*ds_1, ds_2* – are the input Dataset or boolean scalars.

*Constraints*
- If *ds_1* (*ds_2*) is a scalar then it must be a **boolean** data type.
- If *ds_1* (*ds_2*) is a Dataset then it has at least a measure of **boolean** type.
- If both *ds_1* and *ds_2* are Datasets then they must have at least one Identifier Component in common (with the same name and data type).
- If both *ds_1* and *ds_2* are Datasets then either they have one or more boolean measures in common, or at least one of them has only a boolean measure.

*Returns*
If both *ds_1* and *ds_2* are scalar values then the **and** operator returns a boolean value that is the result of the **and** operation.
If either *ds_1* or *ds_2* is a Dataset then the **and** operator returns a Dataset having the following components:
- The superset of the Identifier Componentsof *ds_1* and *ds_2*
- If *ds_1* and *ds_2* have one or more Boolean measures in common (i.e., with the same name) then the resulting Dataset has these common Boolean measures, with the same name, varied on the base of the logical **and**

between the Measure Components of ds_1 and ds_2. Otherwise, if *ds_1* and *ds_2* do not have any measures in common and have only one measure then the resulting Dataset contains a measure named **CONDITION** that contains a Boolean value that is the result of the **and** operation.

The resulting Dataset contains a data point for each pair of data points of *ds_1* and *ds_2* that have the same key (the same values of the Identifier Components).

*Examples*

On scalar

1) If A = True, B = False

C := A **and** B          C = False

On Dataset

2) ds_r:=population.sex="M" **and** population.age="Y15-64"

| population | | | | |
|---|---|---|---|---|
| **SEX** | **AGE** | **GEO** | **TIME** | **VALUE** |
| M | Y_LT15 | BE | 2013 | 970428 |
| M | Y15-64 | BE | 2013 | 3678355 |
| M | Y_GE65 | BE | 2013 | 838653 |
| F | Y_LT15 | BE | 2013 | 927644 |
| F | Y15-64 | BE | 2013 | 3625561 |
| F | Y_GE65 | BE | 2013 | 1121001 |
| M | Y_LT15 | UK | 2013 | 5757444 |
| M | Y15-64 | UK | 2013 | 20748657 |
| M | Y_GE65 | UK | 2013 | 4917238 |
| F | Y_LT15 | UK | 2013 | 5488356 |
| F | Y15-64 | UK | 2013 | 20915924 |
| F | Y_GE65 | UK | 2013 | 6068452 |

| ds_r | | | | |
|---|---|---|---|---|
| **SEX** | **AGE** | **GEO** | **TIME** | **CONDITION** |
| M | Y_LT15 | BE | 2013 | false |
| M | Y15-64 | BE | 2013 | true |
| M | Y_GE65 | BE | 2013 | false |
| F | Y_LT15 | BE | 2013 | false |
| F | Y15-64 | BE | 2013 | false |
| F | Y_GE65 | BE | 2013 | false |
| M | Y_LT15 | UK | 2013 | false |
| M | Y15-64 | UK | 2013 | true |
| M | Y_GE65 | UK | 2013 | false |
| F | Y_LT15 | UK | 2013 | false |
| F | Y15-64 | UK | 2013 | false |
| F | Y_GE65 | UK | 2013 | false |

## or

*Semantics*

The **or** operator calculates the logical OR

*Syntax*

*ds_1* **or** *ds_2*

*Parameters*

*ds_1, ds_2* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as boolean}+ {attribute <IDENT> as scalar-type}*|boolean]

*ds_1, ds_2* – are the input Dataset or boolean scalars.

*Constraints*

- If *ds_1* (*ds_2*) is a scalar then it must be a **boolean** data type.
- If *ds_1* (*ds_2*) is a Dataset then it has at least a measure of **boolean** type.
- If both *ds_1* and *ds_2* are Datasets then they must have at least one Identifier Component in common (with the same name and data type).
- If both *ds_1* and *ds_2* are Datasets then either they have one or more boolean Measures in common, or at least one of them has only a boolean measure.

*Returns*

If both *ds_1* and *ds_2* are scalar values then the **or** operator returns a boolean value that is the result of the **or** operation.

If either *ds_1* or *ds_2* is a Dataset then the **or** operator returns a Dataset having the following components:

- The superset of the Identifier Components of *ds_1* and *ds_2*
- If *ds_1* and *ds_2* have one or more boolean Measures in common (i.e., with the same name) then the resulting Dataset has these common boolean Measures, with the same name, varied on the base of the logical **or** between the Measure Components of ds_1 and ds_2. Otherwise, if *ds_1* and *ds_2* do not have any Measures in common and have only one Measure then the resulting Dataset contains a Measure named **CONDITION** that contains a boolean value that is the result of the **or** operation.

The resulting Dataset contains a data point for each pair of data points of *ds_1* and *ds_2* that have the same key (the same values of the Identifier Components).

*Examples*

On scalar

1) If A = True, B = False

C := A **or** B          C = True

On Dataset

2) ds_r:=population.sex=”M” **OR** population.age_group=”Y15-64”

| population | | | | |
|---|---|---|---|---|
| **SEX** | **AGE** | **GEO** | **TIME** | **CONDITION** |
| M | Y_LT15 | BE | 2013 | 970428 |
| M | Y15-64 | BE | 2013 | 3678355 |
| M | Y_GE65 | BE | 2013 | 838653 |
| F | Y_LT15 | BE | 2013 | 927644 |
| F | Y15-64 | BE | 2013 | 3625561 |
| F | Y_GE65 | BE | 2013 | 1121001 |
| M | Y_LT15 | UK | 2013 | 5757444 |
| M | Y15-64 | UK | 2013 | 20748657 |
| M | Y_GE65 | UK | 2013 | 4917238 |
| F | Y_LT15 | UK | 2013 | 5488356 |

| | | | | |
|---|---|---|---|---|
| F | Y15-64 | UK | 2013 | 20915924 |
| F | Y_GE65 | UK | 2013 | 6068452 |

| DS_or | | | | |
|---|---|---|---|---|
| **SEX** | **AGE** | **GEO** | **TIME** | **CONDITION** |
| M | Y_LT15 | BE | 2013 | true |
| M | Y15-64 | BE | 2013 | true |
| M | Y_GE65 | BE | 2013 | true |
| F | Y_LT15 | BE | 2013 | false |
| F | Y15-64 | BE | 2013 | true |
| F | Y_GE65 | BE | 2013 | false |
| M | Y_LT15 | UK | 2013 | true |
| M | Y15-64 | UK | 2013 | true |
| M | Y_GE65 | UK | 2013 | true |
| F | Y_LT15 | UK | 2013 | false |
| F | Y15-64 | UK | 2013 | true |
| F | Y_GE65 | UK | 2013 | true |

## xor

*Semantics*

The **xor** operator calculates the logical XOR

*Syntax*

*ds_1* **xor** *ds_2*

*Parameters*

*ds_1, ds_2* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as boolean}+
{attribute <IDENT> as scalar-typer }*|boolean]

*ds_1, ds_2* – are the input Dataset or boolean scalars.

*Constraints*

- If *ds_1* (*ds_2*) is a scalar then it must be a **boolean** data type.
- If *ds_1* (*ds_2*) is a Dataset then it has at least a Measure of **boolean** type.
- If both *ds_1* and *ds_2* are Datasets then they must have at least one Identifier Componentin common (with the same name and data type).
- If both *ds_1* and *ds_2* are Datasets then either they have one or more boolean Measures in common, or at least one of them has only a Boolean Measure.

*Returns*

If both *ds_1* and *ds_2* are scalar values then the **xor** operator returns a boolean value that is the result of the **xor** operation.
If either *ds_1* or *ds_2* is a Dataset then the **xor** operator returns a Dataset having the following components:

- The superset of the identifier components of *ds_1* and *ds_2*
- If *ds_1* and *ds_2* have one or more boolean Measures in common (i.e., with the same name) then the resulting Dataset has these common boolean Measures, with the same name, varied on the base of the logical **xor**

between the Measure Components of ds_1 and ds_2. Otherwise, if *ds_1* and *ds_2* do not have any Measures in common and have only one Measure then the resulting Dataset contains a Measure named **CONDITION** that contains a boolean value that is the result of the **xor** operation.

The resulting Dataset contains a data point for each pair of data points of *ds_1* and *ds_2* that have the same key (the same values of the Identifier Components).

*Examples*

On scalar

1) If A = True, B = False

C := A **or** B C = True

On Dataset

2) DS_xor:=population.sex=”M” **xor** population.age_group=”Y15-64”

| DS_xor | | | | |
|---|---|---|---|---|
| SEX | AGE | GEO | TIME | CONDITION |
| M | Y_LT15 | BE | 2013 | 970428 |
| M | Y15-64 | BE | 2013 | 3678355 |
| M | Y_GE65 | BE | 2013 | 838653 |
| F | Y_LT15 | BE | 2013 | 927644 |
| F | Y15-64 | BE | 2013 | 3625561 |
| F | Y_GE65 | BE | 2013 | 1121001 |
| M | Y_LT15 | UK | 2013 | 5757444 |
| M | Y15-64 | UK | 2013 | 20748657 |
| M | Y_GE65 | UK | 2013 | 4917238 |
| F | Y_LT15 | UK | 2013 | 5488356 |
| F | Y15-64 | UK | 2013 | 20915924 |
| F | Y_GE65 | UK | 2013 | 6068452 |

| DS_xor | | | | |
|---|---|---|---|---|
| SEX | AGE | GEO | TIME | CONDITION |
| M | Y_LT15 | BE | 2013 | true |
| M | Y15-64 | BE | 2013 | false |
| M | Y_GE65 | BE | 2013 | true |
| F | Y_LT15 | BE | 2013 | false |
| F | Y15-64 | BE | 2013 | true |
| F | Y_GE65 | BE | 2013 | false |
| M | Y_LT15 | UK | 2013 | true |
| M | Y15-64 | UK | 2013 | false |
| M | Y_GE65 | UK | 2013 | true |
| F | Y_LT15 | UK | 2013 | false |
| F | Y15-64 | UK | 2013 | true |
| F | Y_GE65 | UK | 2013 | false |

# not

*Semantics*

The **not** operator calculates the logical negation of a boolean condition

*Syntax*

**not** *ds_1*

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as string-literal}+ {attribute <IDENT> as scalar-type}* | boolean]

*ds* – is a Dataset expression or a string

*Constraints*

If *ds* is a scalar then it must be a **boolean** type.
If *ds* is a Dataset then it must have at least a Measure of type **boolean**.

*Returns*

If *ds* is a scalar then **not** returns the logical negation of *ds*.
If *ds* is a Dataset and has N boolean Measures then **not** returns a Dataset having the Identifier Components of *ds* and N numeric Measures with the same name of the boolean Measures of *ds* and containing the logical negation of the corresponding Measures.

*Examples*

On scalar

1) If A = True

B := **not** A          B = False

On Dataset

2) ds_r:=**not** population.sex="M"

| population | | | | |
|---|---|---|---|---|
| **SEX** | **AGE** | **GEO** | **TIME** | **CONDITION** |
| M | Y_LT15 | BE | 2013 | 970428 |
| M | Y15-64 | BE | 2013 | 3678355 |
| M | Y_GE65 | BE | 2013 | 838653 |
| F | Y_LT15 | BE | 2013 | 927644 |
| F | Y15-64 | BE | 2013 | 3625561 |
| F | Y_GE65 | BE | 2013 | 1121001 |
| M | Y_LT15 | UK | 2013 | 5757444 |
| M | Y15-64 | UK | 2013 | 20748657 |
| M | Y_GE65 | UK | 2013 | 4917238 |
| F | Y_LT15 | UK | 2013 | 5488356 |
| F | Y15-64 | UK | 2013 | 20915924 |
| F | Y_GE65 | UK | 2013 | 6068452 |

| ds_r | | | | |
|---|---|---|---|---|
| **SEX** | **AGE** | **GEO** | **TIME** | **CONDITION** |
| M | Y_LT15 | BE | 2013 | false |

| | | | | |
|---|---|---|---|---|
| M | Y15-64 | BE | 2013 | false |
| M | Y_GE65 | BE | 2013 | false |
| F | Y_LT15 | BE | 2013 | true |
| F | Y15-64 | BE | 2013 | true |
| F | Y_GE65 | BE | 2013 | true |
| M | Y_LT15 | UK | 2013 | false |
| M | Y15-64 | UK | 2013 | false |
| M | Y_GE65 | UK | 2013 | false |
| F | Y_LT15 | UK | 2013 | true |
| F | Y15-64 | UK | 2013 | true |
| F | Y_GE65 | UK | 2013 | true |

# VTL-ML - Date operators and functions

## extract

*Semantics*

The operator **extract** returns an integer that is part of a given date, based on the value assumed by the *part* parameter.

*Syntax*

**extract(** *ds, part* **)**

*Parameters*

*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as date}+
{attribute <IDENT> as scalar-type}*| date]
*part* : string

*ds* – is the input Dataset or date.
*part* – is the part of date (year, month or day) to extract.

*Constraints*

- *part* can assume a restricted number of values (“A”, "S", "Q", “M” , "W", “D”).

If *ds* is a Dataset, it must have only date Measure Component.

*Returns*

If *ds* is a date value then the **extract** operator returns an integer value that is the part of *ds* specified in the *part* parameter.
If *ds* is a Dataset then the **extract** operator returns a Dataset having all the Identifier, Measure and Attribute Components of *ds*, where the Measure Components change the data in type (from date to integer) and assume the values of part of the dates (on the base of the *part*) in the input Measure Components

*Examples*

On date
1) If A = 28/02/2016
B := **extract** (A, “Y”) B = 2016
B := **extract** (A, “M”) B = 2

On Dataset
2) ds_2:= **extract (**ds_1, “Y”)

| ds_1 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 2015/12/10 |
| 2 | B | 2016/06/11 |
| 3 | C | 2015/12/10 |
| 4 | E | 2013/06/11 |

| ds_2 | | |
|---|---|---|
| **K1** | **K2** | **M1** |
| 1 | A | 2015 |
| 2 | B | 2016 |
| 3 | C | 2015 |
| 4 | E | 2013 |

# string from date

*Semantics*
The operator **string_from_date** converts a date value into a string.

*Syntax*
**string_from_date(** *ds, format* **)**

*Parameters*
*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as date}+
{attribute <IDENT> as scalar-type}*|date]
*format* : string-literal

date – is the input Dataset or date.
format – is the format of the resulting string.

*Constraints*
- If *ds* is a Dataset then it must have only one date Measure Component,
- *format* must respect one of the following patterns:

| Format | Frequency | Example | Frequency |
|---|---|---|---|
| YYYY | | 2000 | Annual |
| YYYYSN | S | 2000S1 | Semestrial |
| YYYYQN | Q | 2000Q1 | Quarterly |
| YYYYMNN | M | 2000M01 | Monthly |
| YYYYDNNNN | D | 2000D0101 | Daily |
| YYYYA | A | 2000A | Annual |
| YYYYSN | S | 2000S1 | Semestrial |
| YYYY-QN | Q | 2000-Q1 | Quarterly |
| YYYY-NN | M | 2000-01 | Monthly |
| YYYY-NN-NN | D, M, Q or A | 2000-01-01 | Daily, Monthly, Quarterly or Annual |

*Returns*
If *ds* is a date then the operator returns a string representation of the input date, based on the chosen format.
If *ds* is a Dataset then the operatorreturns a Dataset having all the Identifier, Measure and Attribute Components of the *ds*, where the Measure Components change in the data type (from date to string-literal) and assume the values of the string representations (on the base of the *format*) of the dates in the input Measure Components.

*Semantic specification*
If the format does not conform to any of the formats expressed in the constraints section, then a runtime exception is raised.

*Examples*
On date
1) If A = 28/02/2016
B := **string_from_date (**A, YYYY-MM**)** B = "2016-02"

On Dataset
2) ds_2:= **string_from_date (**ds_1, “YYYY-MM”**)**

| ds_1 |
|---|

| K1 | K2 | M1 |
|---|---|---|
| 1 | A | 2015/12/10 |
| 2 | B | 2015/06/11 |
| 3 | C | 2015/12/10 |
| 4 | E | 2015/06/11 |

| ds_2 | | |
|---|---|---|
| K1 | K2 | M1 |
| 1 | A | “2015-12” |
| 2 | B | “2015-06” |
| 3 | C | “2015-12” |
| 4 | E | ”2015-06” |

## current_date

*Semantic*

The operator **current_date** returns the current date.

*Syntax*

***current_date()***

*Parameters*

*None*

*Constraints*

None

*Returns*

A Dataset having only one date Measure Component, with only one single Data Point representing the current date.

# VTL-ML - Set functions

## union

*Semantics*

The operator **union** takes as input a list of Datasets and returns a single Dataset containing all the Data Points, without duplicates, that appear in any of them.

*Syntax*

**union (** *ds* {, *ds*}* {,**dedup(**consResFunction**)** }?**)**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}*
{attribute <IDENT> as scalar-type}+

*consResFunctions* : list<component-ref * (t*t) -> t > (t is the type of the referred Component)

- *ds* – are the input Datasets.
- *consResFunction* is a list of functions used to solve conflicts caused by the presence of Data Points with the same values for the Identifier Components.

*Constraints*

All the *ds* Datasets must have the same Identifier and Measure Components, in name and type (static).

*Returns*

The operator allows to eliminate duplicates through *consResFunction.* If the resulting set of data contains duplicates then **union** generates a run-time error.

*Semantic specification*

The operator takes as input a list of Datasets and returns a Dataset with the same structure as the input one and containing all the Data Points from every *ds* without duplicates. The consResFunction allows the user to specify a strategy to eliminate duplicates. In particular, for any single n-uple of duplicate Data Points, the function is applied recursively so as to reduce the duplicates to one single Data Point.If only a Dataset is specified, then it is returned unchanged.

*Examples*

1) ds_r := **union(**total_population1, total_population2**)**

| total_population1 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 5 |
| 2012 | Greece | Total | Total | 2 |
| 2012 | France | Total | Total | 3 |
| 2012 | Malta | Total | Total | 7 |
| 2012 | Finland | Total | Total | 9 |
| 2012 | Switzerland | Total | Total | 12 |

| Total_population2 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Netherlands | Total | Total | 23 |

| | | | | |
|---|---|---|---|---|
| 2012 | Greece | Total | Total | 2 |
| 2012 | Spain | Total | Total | 5 |
| 2012 | Iceland | Total | Total | 1 |

| ds_r | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 5 |
| 2012 | Greece | Total | Total | 2 |
| 2012 | France | Total | Total | 3 |
| 2012 | Malta | Total | Total | 7 |
| 2012 | Finland | Total | Total | 9 |
| 2012 | Switzerland | Total | Total | 12 |
| 2012 | Netherlands | Total | Total | 23 |
| 2012 | Spain | Total | Total | 5 |
| 2012 | Iceland | Total | Total | 1 |

2) ds_r := **union(**total_population1, total_population2**)**

| total_population1 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 1 |
| 2012 | Greece | Total | Total | 2 |
| 2012 | France | Total | Total | 3 |
| 2012 | Malta | Total | Total | 4 |
| 2012 | Finland | Total | Total | 5 |
| 2012 | Switzerland | Total | Total | 6 |

| total_population2 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2011 | Belgium | Total | Total | 10 |
| 2012 | Greece | Total | Total | 20 |
| 2012 | France | Total | Total | 30 |
| 2012 | Malta | Total | Total | 40 |
| 2012 | Finland | Total | Total | 50 |
| 2012 | Switzerland | Total | Total | 60 |

| total_population1 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 1 |
| 2012 | Greece | Total | Total | 2 |

| | | | | |
|---|---|---|---|---|
| 2012 | France | Total | Total | 3 |
| 2012 | Malta | Total | Total | 4 |
| 2012 | Finland | Total | Total | 5 |
| 2012 | Switzerland | Total | Total | 6 |
| 2011 | Belgium | Total | Total | 10 |

3) total_population := **union (**total_population1, total_population2**)**

| **total_population1** | | |
|---|---|---|
| **TIME** | **GEO** | **POPULATION** |
| 2012 | Belgium | 5 |
| 2012 | Greece | 2 |
| 2012 | France | 3 |
| 2012 | Malta | 7 |
| 2012 | Finland | 9 |
| 2012 | Switzerland | 12 |

| **total_population2** | | |
|---|---|---|
| **TIME** | **GEO** | **POPULATION** |
| 2012 | Netherlands | 23 |
| 2012 | Greece | 2 |
| 2012 | Spain | 5 |
| 2012 | Iceland | 1 |

| **total_population** | | |
|---|---|---|
| **TIME** | **GEO** | **POPULATION** |
| 2012 | Belgium | 5 |
| 2012 | Greece | 2 |
| 2012 | France | 3 |
| 2012 | Malta | 7 |
| 2012 | Finland | 9 |
| 2012 | Switzerland | 12 |
| 2012 | Netherlands | 23 |
| 2012 | Spain | 5 |
| 2012 | Iceland | 1 |

4) time_geo := **union (**time_geo1, time_geo2**)**

| **time_geo1** | |
|---|---|
| **TIME** | **GEO** |
| 2012 | Belgium |
| 2012 | Greece |

| | |
|---|---|
| 2012 | France |
| 2012 | Malta |
| 2012 | Finland |
| 2012 | Switzerland |

| time_geo2 | |
|---|---|
| **TIME** | **GEO** |
| 2012 | Netherlands |
| 2012 | Greece |
| 2012 | Spain |
| 2012 | Iceland |

| time_geo | |
|---|---|
| **TIME** | **GEO** |
| 2012 | Belgium |
| 2012 | Greece |
| 2012 | France |
| 2012 | Malta |
| 2012 | Finland |
| 2012 | Switzerland |
| 2012 | Netherlands |
| 2012 | Spain |
| 2012 | Iceland |

## intersect

*Semantics*

The operator **intersect** takes as input Datasets and returns another Dataset with the intersection of the input Datasets.

*Syntax*

**intersect (** *ds* {, *ds*}* {,**dedup(**consResFunction**)** }?**)**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}*
{attribute <IDENT> as scalar-type}+

*consResFunctions* : list<component-ref * (t*t) -> t > (t is the type of the referred Component)

- *ds* – are the input Datasets.
- *consResFunction* is a list of functions used to solve conflicts caused by the presence of Data Points with the same values for the Identifier Components.

*Constraints*

All the Datasets *ds* must have the same Identifier and Measure Components, in name and type (static).

*Returns*

A Dataset having the same Identifier, Measure and Attribute Components of the input ones, containing all the Data Points that are present in every *ds*.

*Semantic specification*

The operator takes as input Datasets and returns another one Dataset with the same structure of the input ones containing all the Data Points that are present in every *ds*, which is their intersection. If two Data Points appear in all the input Datasets, but with different values for the Measure Components, then the values for the Measures are determined by combining the input ones with a *consResFunction* that solves the conflicts.

*Examples*

d_r := **intersect(**total_population1, total_population2**)**

| total_population1 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 1 |
| 2012 | Greece | Total | Total | 2 |
| 2012 | France | Total | Total | 3 |
| 2012 | Malta | Total | Total | 4 |
| 2012 | Finland | Total | Total | 5 |
| 2012 | Switzerland | Total | Total | 6 |

| total_population2 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2011 | Belgium | Total | Total | 10 |
| 2012 | Greece | Total | Total | 2 |
| 2011 | France | Total | Total | 30 |
| 2011 | Malta | Total | Total | 40 |
| 2011 | Finland | Total | Total | 50 |
| 2011 | Switzerland | Total | Total | 60 |

| d_r | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Greece | Total | Total | 2 |

## symdiff

*Semantics*

The operator **symdiff** takes as input two Datasets and returns another Dataset with the symmetric difference of the input Datasets.

*Syntax*

**symdiff (** *ds_1*, *ds_2* **)**

*Parameters*

*ds_1, ds_2* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}*
{attribute <IDENT> as scalar-type}+

- *ds_1* – is the first input Dataset.
- *ds_2* – is the second input Dataset.

*Constraints*
*ds_1* and *ds_2* must have the same Identifier and Measure Components in name and type (static).

*Returns*
A Dataset having the same Identifier, Measure and Attribute Components of the input ones, containing all the Data Points that are present either in *ds_1* or in *ds_2* but not in both.

*Semantic specification*
The operator takes as input two Datasets and returns another one Dataset with the same structure of the input ones containing all the Data Points that are present either in *ds_1* or in *ds_2* but not in both.

*Examples*
d_r := **symdiff(**total_population1, total_population2**)**

| total_population1 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 1 |
| 2012 | Greece | Total | Total | 2 |
| 2012 | France | Total | Total | 3 |
| 2012 | Malta | Total | Total | 4 |
| 2012 | Finland | Total | Total | 5 |
| 2012 | Switzerland | Total | Total | 6 |

| total_population2 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2011 | Belgium | Total | Total | 1 |
| 2012 | Greece | Total | Total | 2 |
| 2012 | France | Total | Total | 3 |
| 2012 | Malta | Total | Total | 4 |
| 2012 | Finland | Total | Total | 5 |
| 2012 | Switzerland | Total | Total | 6 |

| d_r | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 1 |
| 2011 | Belgium | Total | Total | 1 |

## setdiff

*Semantics*
The operator **setdiff** takes as input two Datasets and returns another Dataset with the difference of the input Datasets.

*Syntax*
**setdiff (** *ds_1* **,** *ds_2* **)**

*Parameters*
*ds_1, ds_2* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}*

{attribute <IDENT> as scalar-type}+

- *ds_1* – is the first input Dataset.
- *ds_2* – is the second input Dataset.

*Constraints*

*ds_1* and *ds_2* must have the same Identifier and Measure Components in name and type (static).

*Returns*

A Dataset having the same Identifier, Measure and Attribute Components of the input ones, containing all the Data Points that are present in ds_1 but not in ds_2.

*Semantic specification*

The operator takes as input two Datasets and returns another Dataset with the same structure as the input ones containing all the Data Points that are present in either *ds_1* but not in *ds_2*, which is their difference.

*Examples*

1) d_r := **setdiff** ( total_population1,total_population2)

| total_population1 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 10 |
| 2012 | Greece | Total | Total | 20 |
| 2012 | France | Total | Total | 30 |
| 2012 | Malta | Total | Total | 40 |
| 2012 | Finland | Total | Total | 50 |
| 2012 | Switzerland | Total | Total | 60 |

| total_population2 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2011 | Belgium | Total | Total | 10 |
| 2012 | Greece | Total | Total | 20 |
| 2012 | France | Total | Total | 30 |
| 2012 | Malta | Total | Total | 40 |
| 2012 | Finland | Total | Total | 50 |
| 2012 | Switzerland | Total | Total | 60 |

| d_r | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 10 |

2) DatasetC := **setdiff** (DatasetA ,DatasetB)

| Dataset A | | | |
|---|---|---|---|
| **COUNTRY** | **SEX** | **YEAR** | **VALUE** |
| FR | Males | 2011 | 7 |
| FR | Females | 2011 | 10 |
| FR | Total | 2011 | 12 |

| Dataset B | | | |
|---|---|---|---|
| COUNTRY | SEX | YEAR | VALUE |
| FR | Males | 2011 | 7 |
| FR | Females | 2011 | 10 |

| Dataset C | | | |
|---|---|---|---|
| COUNTRY | SEX | YEAR | VALUE |
| FR | Total | 2011 | 12 |

## subscript

*Semantics*

The operator **subscript** takes as input a Dataset and a sequence of Identifier Components with their respective values, and returns another Dataset having only the data points that contains the values specified in the subscript for the respective Identifier Component.

*Syntax*

*ds* **[** *comp* = *comp_value1* { **,** *comp* **=** *comp_value2* } * **]**

*Parameters*

- *ds* – is the input Dataset
- *comp* – Dataset component-ref
- *comp_value1, comp_value2* – is a valid value for component

*Constraints*

- *comp* must be a valid Identifier of *ds* component.
- *comp_value1, comp_value2* must be a valid value for the related component.

*Returns*

A Dataset having the same Measure and Attribute Components as the input one, and all the Identifier Components that are not specified as parameters (*comp*). The Data points of the returned Dataset are all those of *ds* whose values having for the subscripted identifier component(s) concide with the values specified in the subscript.

*Semantic specification*

This operator removes identifiers components of the Dataset performing before a filter over the components values specified in the subscript. This avoids inconsistency on the returned Dataset.

*Examples*

1) ds_2 := ds_1 **[** time = 2010, ref_area = EU25 **]**

| ds_1 | | | | |
|---|---|---|---|---|
| TIME | REF_AREA | PARTNER | OBS_VALUE | OBS_STATUS |
| 2010 | EU25 | CA | 20 | D |
| 2010 | BG | CA | 1 | P |
| 2010 | RO | CA | 1 | P |
| 2010 | EU27 | CA | 23 | P |

| ds_2 | | |
|---|---|---|
| **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| CA | 20 | D |

2) ds_2 := ds_1 **[**time = 2010, ref_area = EU25, partner = CA **]**

| ds_2 | |
|---|---|
| **OBS_VALUE** | **OBS_STATUS** |
| 20 | D |

3) ds_2 := ds_1 **[**ref_area = EU25 **]** + ds_1**[** ref_area = BG **]** + ds_1 **[** ref_area = RO **]**

| ds_2 | | |
|---|---|---|
| **TIME** | **PARTNER** | **OBS_VALUE** |
| 2010 | CA | 22 |

4) ds_2 := ds_1 **[** time = 2010, ref_area = EU25 **]**

| ds_1 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 20 | D |
| 2010 | EU25 | NF | 1 | P |
| 2010 | RO | CA | 1 | P |
| 2010 | EU27 | CA | 23 | P |

| ds_2 | | |
|---|---|---|
| PARTNER | OBS_VALUE | OBS_STATUS |
| CA | 20 | D |
| NF | 1 | P |

## transcode

*Semantics*

The **transcode** operator recodes the identifiers values using a map Dataset or a mapping object.

*Syntax*

**transcode(***ds.comp*, [*ds_map*| *mapping*]**)**

*Parameters*

*ds.comp* : Component-ref
*ds* : dataset {identifier <IDENT> as scalar-type}+
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*
*ds_map* : dataset {identifier **MAPS_FROM** as scalar-type; } {measure **MAPS_TO** as scalar-type; }

- *ds.comp* – is a valid Identifier Component of the Dataset.

- *ds_map* – is the Dataset that the defines the mapping. It has an Identifier Component, **MAPS_FROM**, that specifies the values to be transformed and a Measure Component, **MAPS_TO**, specifying the target value for each of them.
- *mapping* – a mapping, persistent object created with define mapping

### *Constraints*

The following conditions guarantee that the resulting Dataset does not have duplicates:

- All the values of the Measure Component **MAPS_TO** must be distinct.
- For each distinct value of the Identifier Component to be recoded, there is a value (and only one) in the Identifier Component **MAPS_FROM** in *ds_map* or in the *mapping* object

### *Returns*

A Dataset that has the same Identifier, Measure and Attribute Components as the input one. The values of the Identifier Component are recoded into the corresponding values in the **MAPS_TO** Measure Component of the Dataset *ds_map*.

### *Semantic specification*

This operator allows to transform an input Dataset by mapping the values of one Identifier Component into corresponding values, as specified by a mapping Dataset. Since the mapping Dataset is guaranteed to have one distinct target value for each input one, and the input values appear only once, the resulting Dataset will contain no duplicates.
All the Data Points of *ds* are also present in the result and the values of the Identifier Component *ds.comp* are modified as follows. For each data points of the Dataset, the value *v* of *ds.comp* is replaced by the value included in the *ds_map* or in the *mapping* corresponding to *v*.

### *Examples*

ds_2 := **transcode(** ds, ds_map, REF_AREA **)**

| ds_map | |
|---|---|
| MAPS_FROM | MAPS_TO |
| LU | LUX |
| BE | BEL |
| IT | ITA |

| ds | |
|---|---|
| REF_AREA | VALUE |
| LU | 10 |
| BE | 11 |
| IT | 13 |

| ds_2 | |
|---|---|
| REF_AREA | VALUE |
| LUX | 10 |
| BEL | 11 |
| ITA | 13 |

## aggregate

### *Semantics*

The operator **aggregate** takes as input a Dataset and returns a new Dataset with the data aggregated based on the rules and Boolean conditions specified in the hierarchical ruleset.

*Syntax*

**aggregate (***ds***,** *hr* **,** { [**total** | **partial**] } **,** { [**return aggregates** | **return all data points**] } **);**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as numeric}+
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*

- *ds* – is the input Dataset to aggregate.
- *hr* – is the hierarchical ruleset (see define hierarchical ruleset) where the rules and the conditions to perform the aggregate operation are defined.
- *total* – a keyword to specify that the aggregation is performed only when all the elements in the right-hand elements of the aggregation conditions in vr are not NULL (default behaviour).
- *partial* – a keyword to specify that the aggregation is performed when at least one element of the right-hand side of the aggregation conditions in vr is not NULL and, in this case, all the NULLs are treated as zero (that is, ignored in the summation).
- *return aggregates* – a keyword to specify that the output Dataset contains only the data points resulting from aggregations (default behaviour).
- *return all data points* – a keyword to specify that the output Dataset contains data points resulting from aggregations as well as the data points of the input Dataset ds.

*Constraints*

- *ds* must have at least one numeric Measure.
- *hr* must be defined for calculation purposes (hence following the respective constraints).

*Returns*

A Dataset with all the Identifier and Measure Components of *ds*, with the data aggregated on the basis of the rules and Boolean conditions specified in the hierarchical (vertical) Ruleset *hr*.

*Semantic specification*

The aggregate operator takes as input a Dataset, with at least a numeric Measure Component, and a hierarchical ruleset and returns a new Dataset, with the data aggregated based on the rules and Boolean conditions specified in the ruleset.
The operator computes the numeric Measure Components associated to the aggregates defined in the left side of the rules in *hr*. The aggregation is prerformed computing all aggregates in a single operation according to a bottom-up calculation.
The rules are executed in an appropriate order. In practice, if a rule in the ruleset depends on another one, the latter is evaluated before, and its output exploited by the former. The functional constraints ensure that each aggregate is calculated once.
By default, the aggregation is performed only when all element of the right side of an aggregation rule in the hierarchical (vertical) Ruleset of input *hr* are not NULL in the input Dataset *ds* (**total** clause). By specifying the **partial** clause the aggregation is performed either if there are NULL values.
The Dataset's data points that are not implied in the aggregation are not shown in the resulting Dataset, essentially the data points containing values that are not involved in the aggregation will be lost (**return aggregates** clause). Specifying the **return all data points** clause, the returned Dataset will contain also the disaggregated data points of the input Dataset *ds.*

*Examples*

In this example an aggregation is performed using the following hierarchical ruleset.

```
define hierarchical ruleset  hr_ref_area ( condition ( time ) rule ( ref_area ) ) is
    EU15 = AT + BE + LU + DE + ES + FI + FR + EL + IE + IT + NL + PT + DK + UK + SE ;
    EU25 = EU15 + CY + CZ + ES + HU + LT + LV + MT + PL + SK + SI ;
    EU27 = EU25 + BG + RO ;
    EU28 = EU27 + HR ;
    when time between 1995 and 2003 then EU = EU15 ;
    when time between 2004 and 2005 then EU = EU25 ;
    when time between 2006 and 2012 then EU = EU27 ;
    when time >= 2013 then EU = EU28
    EEA15 = EU15 + IS + NO + LI ;
```

EEA25 = EU25 + IS + NO + LI ;
EEA27 = EU27 + IS + NO + LI ;
EEA30 = EU27 + IS + NO + LI ;
when time between 1995 and 2003 then EEA = EEA15 ;
when time between 2004 and 2005 then EEA = EEA25 ;
when time between 2006 and 2012 then EEA = EEA27 ;
when time >= 2013 then EEA = EEA30 ;
**end hierarchical ruleset**

The Dataset to aggregate:

| ds_bop | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 20 | D |
| 2010 | BG | CA | 1 | P |
| 2010 | RO | CA | 1 | P |
| 2010 | EU27 | CA | 25 | P |
| 2010 | HR | CA | 2 | P |

1) ds_2 := **aggregate**( ds_bop, hr_ref_area );

| ds_2 | | | |
|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** |
| 2010 | EU27 | CA | 22 |
| 2010 | EU28 | CA | 27 |

2) ds_2 := **aggregate**( ds_bop, hr_ref_area, return all data points );

| ds_2 | | | |
|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** |
| 2010 | EU25 | CA | 20 |
| 2010 | BG | CA | 1 |
| 2010 | RO | CA | 1 |
| 2010 | EU27 | CA | 22 |
| 2010 | HR | CA | 2 |

3) In this example an aggregation is performed using the following hierarchical ruleset.

**define hierarchical ruleset** hr_ref_area (condition ( time ) rule ( ref_area ) ) is
when time = 2010 then EU= IT+BE+LU;
when time = 2010 the AS=IN+CH;
**end hierarchical ruleset**

ds_2 := **aggregate**(ds_bop, hr_ref_area);

| ds_bop | | | | |
| --- | --- | --- | --- | --- |
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | IT | CA | 2 | P |
| 2010 | BE | CA | 1 | P |
| 2010 | LU | CA | 1 | P |
| 2010 | IN | CA | 3 | D |
| 2010 | CH | | 5 | D |

| ds_2 | | | | |
| --- | --- | --- | --- | --- |
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU | CA | 4 | P |
| 2010 | AS | CA | 8 | D |

4) In this example the Italian northern population has been obtained summing the population of nord_east and nord-west (rule 1) and summing the population of all the regions being part of the nord (rule 2).

Hierarchical (vertical) Ruleset:

**define hierarchical ruleset** hr_IT_north_pop (rule ( ref_area ) ) is
    ITCD = ITC + ITD;
    ITCD =ITC1+ITC2+ITC3+ITC4+ITD1+ITD2
**end hierarchical ruleset**

ds_2 := **aggregate**(IT_nord_bop, hr_IT_nord_pop);

| IT_nord_pop | | |
| --- | --- | --- |
| **TIME** | **REF_AREA** | **OBS_VALUE** |
| 2015 | ITCD | 27799803 |
| 2015 | ITC | 16138643 |
| 2015 | ITC1 | 4424467 |
| 2015 | ITC2 | 128298 |
| 2015 | ITC3 | 1583263 |
| 2015 | ITC4 | 10002615 |
| 2015 | ITD | 11661160 |
| 2015 | ITD1 | 518518 |
| 2015 | ITD2 | 537416 |
| 2015 | ITD3 | 4927596 |
| 2015 | ITD4 | 1227122 |
| 2015 | ITD5 | 4450508 |

| ds_2 | | |
| --- | --- | --- |
| **TIME** | **REF_AREA** | **OBS_VALUE** |
| 2015 | ITCD | 27799803 |

# VTL-ML - Statistical functions

## Aggregate functions

*Semantics*

VTL includes a set of statistical functions, that can be used to aggregate data.

*Syntax*

*aggregateFunction* **(** *ds* {, *other_parameters* } **)** { [ **group by** | **along** ] **(** *idComp* {, *idComp* }* **)** }

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type; }+
[ {measure <IDENT> as numeric}+ | {attribute <IDENT> as numeric}+ ]
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*
*idComp* : component-ref

- *aggregateFunction* – is one of the aggregate functions described in the table below.
- *other_parameters* – specific parameters additional to *ds*, related to the function used (see table **List of aggregate functions**)
- *ds* – is the input Dataset to which the aggregate function is applied.
- **group by** – represents the VTL groups data composed by the Identifier Components specified as idComp.
- **along** – represents the VTL groups data composed by the Identifier Components of ds that are not specified as *idComp*. With the **along** clause the same VTL program can be reused for all Datasets that contain the Identifier Components specified in the **along** clause.
- *idComp* – a component identifier of *ds*

*Constraints*

- If *ds* has more than one Measure Component, then a Measure or attribute must be defined using the membership operator.
- *idComp* must be a valid reference to an existing Identifier Component owned by *ds*.

*Returns*

A Dataset having the Identifier Components of *ds* specified in the **group by** clause (or not specified in the **along** clause) and the Measure Components (or the implicit Measure Component deduced in a mono-Measure Dataset), with the data aggregated on the basis of the specific aggregate function and the partitions defined by **group by** or **along**.

*Semantic specification*

An aggregate function groups together, evaluating a value that is specific for each aggregate function, the values of multiple data points having the same values of the specified Identifier Components *idComp*.
The operator takes as input a Dataset, a Measure Component (specified with the membership operator on *ds,* or implicitly selected if *ds* has only one Measure component) on which the aggregate function will compute the result, and a sequence of Identifier Components *idComp* that will be used for the partitioning (the aggregate function is then applied separately for each partition). It returns another Dataset having the Identifier Components of *ds,* specified in the **group by** or **along,** and the Measure Component used for the aggregation. The other Identifier Components are removed from the resulting Dataset.
If neither a **group by** or **along** clause is specified, then the aggregate function returns a single Data Point that has zero Identifier Components and only one Measure Component that is the one specified for the aggregation with the membership operator (or deduced from *ds*).
Most of the aggregate functions can be also used as analytic functions (with a different syntax): See analytic functions.

| List of aggregate functions | |
|---|---|
| **Aggregate function** | **Description** |
| **avg (***ds_1***)** | average value of the not null values of *ds_1* |
| **corr (***ds_1***,** *ds_2***)** | Coefficient of correlation of (*ds_1*, *ds_2*) |
| **covar_pop (** *ds_1***,** *ds_2***)** | population covariance of (*ds_1*, *ds_2*) |
| **covar_samp (***ds_1***,** *ds_2)* | sample covariance of (*ds_1*, *ds_2*) |
| **count (** *ds_1* **)** | number of non-empty data points of *ds_1* |
| **median (** *ds_1* **)** | median value of the not null values of *ds_1* |
| **min (** *ds_1* **)** | minimum value of *ds_1* |
| **max (** *ds_1* **)** | maximum value of *ds_1* |
| **percentile_cont (** *ds_1*, constant **)** order by expression [ **asc** \| **desc** ] | inverse distribution function that assumes a continuous distribution model |
| **percentile_disc (** *ds_1* , constant **)** order by expression [ **asc** \| **desc** ] | inverse distribution function that assumes a discrete distribution model |
| **rank (** *ds_1* **)** | rank of a value in a group of values |
| **regr_slope (** *ds_1, ds_2* **)** | linear regression (slope of the line) |
| **regr_intercept (** *ds_1***,** *ds_2* **)** | linear regression (y-intercept) |
| **regr_count (** *ds_1***,** *ds_2* **)** | linear regression (count non-null number pairs) |
| **regr_r2 (** *ds_1***,** *ds_2* **)** | linear regression (coefficient of determination) |
| **regr_avgx (** *ds_1***,** *ds_2* **)** | linear regression (average of independent variable *ds_2*) |
| **regr_avgy (** *ds_1***,** *ds_2 )* | linear regression (average of dependent variable *ds_1*) |
| **regr_sxx (** *ds_1***,** *ds_2* **)** | linear regression (auxiliary function) |
| **regr_syy (** *ds_1***,** *ds_2* **)** | linear regression (auxiliary function) |
| **regr_sxy (** *ds_1***,** *ds_2* **)** | linear regression (auxiliary function) |
| **stddev_pop (** *ds_1* **)** | population standard deviation of *ds_1* |
| **stddev (** *ds_1* **)** | standard deviation of *ds_1* |
| **sum (** *ds_1* **)** | sum of values of *ds_1* |
| **var_pop (** *ds_1* **)** | population variance of *ds_1* |
| **var_samp (** *ds_1* **)** | sample variance of *ds_1* |
| **variance (** *ds_1* **)** | variance of *ds_1* |

*Examples*

1) ds_agg := **avg (** ds_bop.obs_value **) group by** time

| ds_bop | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 20 | |
| 2010 | BG | CA | 1 | |
| 2010 | RO | CA | 1 | |
| 2010 | EU27 | CA | 23 | |
| 2011 | EU25 | CA | 20 | P |
| 2011 | BG | CA | 1 | P |
| 2011 | RO | CA | -1 | P |
| 2011 | EU27 | CA | 20 | P |
| 2012 | LU | CA | 40 | P |
| 2012 | EU25 | CA | 30 | P |

| ds_agg | |
|---|---|
| **TIME** | **OBS_VALUE** |
| 2010 | 11.25 |
| 2011 | 11.25 |
| 2012 | 30 |

Note: the example above can be rewritten equivalently in the following forms:
ds_agg := **avg (** ds_bop **) along** ref_area, partner
ds_agg := **avg (** bop **) group by** time

2) ds_agg := **avg (** ds_bop.obs_value **) group by** time, ref_area

| ds_agg | | |
|---|---|---|
| **TIME** | **REF_AREA** | **OBS_VALUE** |
| 2010 | EU25 | 20 |
| 2010 | BG | 1 |
| 2010 | RO | 1 |
| 2010 | EU27 | 23 |
| 2011 | EU25 | 20 |
| 2011 | BG | 1 |
| 2011 | RO | -1 |
| 2011 | EU27 | 20 |
| 2012 | LU | 40 |
| 2012 | EU25 | 30 |

3) ds_agg := **avg (** ds_bop **)**

| ds_agg |
| --- |
| **OBS_VALUE** |
| 15.5 |

4) ds_agg := **max (** ds_bop.obs_value **) as** max_value, **min (** ds_bop.obs_value **) as** min_value **group by** time

| ds_agg | | | |
| --- | --- | --- | --- |
| **TIME** | **REF_AREA** | **MIN_VALUE** | **MAX_VALUE** |
| 2010 | EU25 | 23 | 1 |
| 2011 | RO | 20 | -1 |
| 2012 | EU25 | 40 | 30 |

## Time aggregate functions

*Semantics*

The *time aggregate functions* represent a set of statistical functions used to aggregate data on the time dimension.

*Syntax*

*aggregateFunction* **(** *ds*
, *freqFrom*
, *freqTo*
{ , *minPeriods* }
{ , *timePeriodName* }
{ , *timeFormatFrom* }
{ , *timeFormatTo* }
**)**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+
[ {measure <IDENT> as numeric}+ | {attribute <IDENT> as numeric}+ ]
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*

- *aggregateFunction* - is one of the aggregate functions described in the paragraph "Aggregate functions".
- *ds* – is the input Dataset to which the aggregate function is applied.
- *freqFrom* – is the frequency from which the data will be aggregated
- *freqTo* – is the frequency to which the data will be aggregated. *freqTo* must be a lower frequency than *freqFrom*
- *minPeriods* – is an Integer number describing the minimum number of values required to perform the time based aggregation. If *minPeriods* is omitted then the aggregation is performed only if all the periods needed for the aggregation are present.
- *timePeriodName* – is the name of the time period component of the Dataset. Default name is "time".
- *timeFormatFrom* – is the format of the time period relative to *freqFrom*. It must be specified only when *freqFrom* is C (custom).
- *timeFormatTo* – is the format of the time period relative to *freqTo*. It must be specified only when *freqTo* is C (custom).

*Constraints*

- If *ds* has more than one Measure Component, then a Measure or attribute must be specified using the membership operator on ds.
- *timePeriodName* is the name of an Identifier Component owned by *ds*.

- *freqFrom* must be a higher frequency compared with *freqTo*, e.g. *freqFrom* = “M” and *freqTo* = “Q” is correct, while the reverse is not correct.

Possible values for *freqFrom* and *freqTo*:

| *Frequency symbol* | *Frequency* |
|---|---|
| A | Annual |
| S | Semestrial |
| Q | Quarterly |
| M | Monthly |
| W | Weekly |
| D | Daily |

*Returns*

A Dataset with the Identifier Components of ds and a Measure Component containing data of ds aggregated from freqFrom to freqTo. All aggregate functions can be used.

*Semantic specification*

The aggregateFunction first partitions the data set in groups of data having the frequency *freqFrom*, then the data are aggregated to obtain data aggregated having the frequency *freqTo*.
Convert the data contained in *ds* and having the time format specified in *freqFrom* to the format specified in *freqTo.* If *freqFrom* and *freqTo* have different value then an aggregation could occur.

Data in *ds* having the frequency different from *freqFrom* will not be involved by the operator and they will be discarded from the output Dataset.

By default this operator computes the aggregated value only when the values of all sub-periods exist.
To override this behaviour the user can specify a value for the optional argument *minPeriods*: this is a lower bound for the number of periods that must exist in order to produce the aggregation. This means that if the optional parameter *minPeriods* is present, then the aggregation can be performed either if the timeseries Dataset of input does not contain all the necessary DataPoints needed.

Time format:

| *Format* | *Frequency* | *Example* | *Frequency* | *Possible values* |
|---|---|---|---|---|
| yyyy<br>yyyyA | A | 2000 | Annual | yyyy = 1900, ..., 9999 |
| yyyySs<br>yyyy-Ss | S | 2000S1 | Semestrial | s = 1, 2 |
| yyyyQq | Q | 2000Q1 | Quarterly | q = 1, 2, 3, 4 |
| yyyy-Qq | Q | 2000Q1 | Quarterly | q = 1, 2, 3, 4 |
| yyyyMmm<br>yyyy-mm | M | 2000M01 | Monthly | mm = 01, 02, ..., 12 |
| yyyyWnn | W | 2000D0101 | Weekly | nn = 01, 02, ..., 52 |
| yyyyMmmDdd | D | 2000D0101 | Daily | mm = 01, 02, ..., 12<br>dd = 01, 02, ..., 31 |
| yyyy-mm-dd | N | 2000-01-01 | Date (frequency not specified) | mm = 01, 02, ..., 12<br>dd = 01, 02, ..., 31 |
| Combination of yyyy, mm and dd | C | 01-01-2000 | Custom date format | yyyy = 1900, ..., 9999<br>mm = 01, 02, ..., 12<br>dd = 01, 02, ..., 31 |

*Examples*

1) ds_abop := **sum** ( bp_qbop) **time_aggregate** ( "Q", "A" )

| ds_qbop | | | |
|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** |
| 2010Q1 | EU25 | CA | 20 |
| 2010Q2 | EU25 | CA | 20 |
| 2010Q3 | EU25 | CA | 20 |
| 2010Q4 | EU25 | CA | 20 |
| 2010Q1 | EU27 | CA | 30 |
| 2010Q2 | EU27 | CA | 30 |
| 2010Q3 | EU27 | CA | 30 |
| 2010Q4 | EU27 | CA | 30 |
| 2010Q1 | IT | CA | 10 |
| 2010Q2 | IT | CA | 10 |
| 2010Q3 | IT | CA | 10 |
| 2010Q4 | IT | CA | 10 |

The above operation perform a frequency change from quarterly data, where the date has the pattern "YYYYQN", to annual data with the pattern "YYYY". Due to the pattern of the Component TIME, the frequency is deduced and the *frequency_name* parameter can be omitted.

| ds_abop | | | |
|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** |
| 2010 | EU25 | CA | 80 |
| 2011 | EU27 | CA | 120 |
| 2010 | IT | CA | 40 |

| ds_1 | |
|---|---|
| **DATE** | **VALUE** |
| 1939-01-01 | 4400.0 |
| 1939-02-01 | 4400.0 |
| 1939-03-01 | 10600.0 |
| 1939-04-01 | 6800.0 |

2) ds_2 := sum ( ds_1, "M", "A", 4, date, "yyyy-mm-dd", "yyyy-mm-dd" )

Due to the value of *minPeriods* ( 4 ), the annual aggregation is performed when at least 4 data points exist relative to the 1939's months.

| ds_2 | |
|---|---|
| **DATE** | **VALUE** |
| 1939-01-01 | 26200.0 |

# Analytic functions

*Semantics*

The Analytic functions allow to specify operations to be applied on groups of Data Points within a Dataset. The Data Points of the Dataset are first partitioned into groups. Then, in each group, each Data Point is combined with (some of) the others in a customizable way, and for each input Data Point, an output one is produced.
Groups are determined by a list of names of Identifier Components, in such a way that Data Points having the same values for those Identifiers are assigned to the same group.
A **sliding window** is then declared to define for each input Data Point in the window (**current** Data Point), how to produce the corresponding Data Point in the output, by combining its Measure Components with the ones of the other Data Points in the same window. For each window and for each Data Point, the sliding window spans the Data Points to be combined and while moving from the first to the last Data Point in the group, produces the output Data Points. In other words, for each group, the sliding window determines the moving range of Data Points to be combined for each input one. At one extreme, the sliding window can span one single Data Point at a time, implying that Data Points are not combined with the others, producing independent values of the Measures for all the input Data Points; at the other extreme, the sliding window spans the entire window, producing the same value for all the Data Points.
The size of the sliding window is either based on the number Data Points to be included or the specification of a numerical interval.
Finally, there are a number of possible functions that can be applied to combine the Data Points within a sliding window, such as the average value within a sliding window, the cumulative sum, and so on.

*Syntax*

*analyticFunction* **(** *ds* {, *extraParams* } **) over (** { *partitionBy* } { *orderBy* } { *windowingClause* } **)**
    *partitionBy* ::= **partition by** *c_p* { **,** *c_p* } *
    orderBy ::= **order by** *c_o* { **,** *c_o* } * { [ **asc** | **desc** ] }
    *windowingClause* **::=**
    [ **rows** | **range** ]
    **between**
    [ num **preceding** |num **following** | **current row** | **unbounded preceding** | **unbounded following**]
    **and**
    [ num **preceding** |num **following** | **current row** | **unbounded preceding** | **unbounded following** ]

*Parameters*

- *ds* – is the input Dataset.
- *extraParams* – additional parameters (depending on the analytic function).
- *partitionBy* – partitions *ds* into groups based on the value of one or more Identifier Components. If omitted, the function treats all rows of the Dataset as a single partition.
- *c_p* – are valid Identifier Components of *ds* used for the partitioning expressed by *partitionBy*.
- *orderBy* - specifies how Data Points are ordered within each windows (asc is the default).
- *c_o* – are references to valid Components on which the sort is performed within the respective pre-calculated windows.
- The keywords *rows* and *range* define for each row a "sliding window" (set of rows) used for calculating the result of the analytic function. The analytic function is then applied to all the Data Points in the sliding window. The sliding window "slides" through the windows from top to bottom. In particular*:*
    - *rows* – defines a sliding window using a specified number of preceding and following data points relative to the current data point (according to the orderBy clause)
    - *range* – defines a sliding window as a numerical offset relative to the current data point (according to the orderBy clause)
- *unbounded preceding* - indicates that the sliding window starts at the first Data Point of the window.
- *unbounded following* - indicates that the sliding window ends at the last Data Point of the window.
- *current row* – specifies that the window starts or ends at the current Data Point.
- *preceding* – specifies the start point of the sliding window as number of data points preceding the current data point.
- *following* – specifies the end point of the sliding window as number of data points following the current data point.

*Constraints*

- *ds* must have at least one Measure Component (as explicated in the syntax).
- *Analytic functions* cannot be nested.

*Returns*

A Dataset having the same Identifier, Measure and Attribute Components as the input one, where the Measure Components take values depending on the definition of windows, sliding windows and the specific **analytic function**.

*Semantic specification*

The operator takes a Dataset as input, optionally the specification for the partitioning and the internal order of the windows; optionally, it also takes as input the information to define a sliding window. If omitted, there is one single sliding window, coinciding with each of the windows. The operator returns a Dataset with the same Identifier Components, Measure Components and Attribute Components as *ds,* where the value of all the Measure Components take values that depend on the specific **analytic function**, the partitioning criteria and the Data Points in each sliding window.
The functions that can be used as analytic functions are the aggregate functions described in the previous chapter and some specific functions described below.

## first_value

*Semantics*

For each sliding window and for each Measure Component, the operator calculates the first value according to the specified order.

*Examples*

ds := **first_value** ( ds_bop ) **over ( partition by** ref_area , partner **order by** time **)**

| ds_bop | | | |
|---|---|---|---|
| **REF_AREA** | **PARTNER** | **TIME** | **OBS_VALUE** |
| LU | CA | 1993 | 3 |
| LU | CA | 1994 | 4 |
| LU | CA | 1996 | 10 |
| LU | CA | 1997 | 20 |
| LU | US | 1993 | 400 |
| LU | US | 1996 | 500 |
| LU | US | 1997 | 600 |
| LU | WORLD | 1994 | 1000 |

| ds | | | |
|---|---|---|---|
| **REF_AREA** | **PARTNER** | **TIME** | **OBS_VALUE** |
| LU | CA | 1993 | 3 |
| LU | CA | 1994 | 3 |
| LU | CA | 1996 | 3 |
| LU | CA | 1997 | 3 |
| LU | US | 1993 | 400 |
| LU | US | 1996 | 400 |
| LU | US | 1997 | 400 |
| LU | WORLD | 1994 | 1 000 |

## lag lead

*Semantics*

The operator swaps the values of all Measure Components of the current Data Point with the ones of the corresponding Measure Components of the Data Point that is referred to by the offset. If the offset exceeds the boundaries of the sliding window, the default value is used for the swap. If omitted, 0 is implied as the default.

*Parameters*

This analytic function takes as input the following extra parameters:

- *offset* – it allows to individuate a Data Point by specifying the relative position from the current Data Point as an offset, negative if moving towards from the beginning to the end of the define ordering, positive if moving from the end to the beginning of the defined order.
- *default* - the value that a Data Point has to take if the Data Point, whose position is calculated using the offset, is NULL.

*Examples*

ds := **lag (** ds_bop , -1, -100 **) over ( partition by** ref_area , partner **order by** time **)**

| ds_bop | | | |
|---|---|---|---|
| **REF_AREA** | **PARTNER** | **TIME** | **OBS_VALUE** |
| LU | CA | 1993 | 3 |
| LU | CA | 1994 | 4 |
| LU | CA | 1996 | 10 |
| LU | CA | 1997 | 20 |
| LU | US | 1993 | 400 |
| LU | US | 1996 | 500 |
| LU | US | 1997 | 600 |
| LU | WORLD | 1994 | 1000 |

| ds | | | |
|---|---|---|---|
| **REF_AREA** | **PARTNER** | **TIME** | **OBS_VALUE** |
| LU | CA | 1993 | -100 |
| LU | CA | 1994 | 3 |
| LU | CA | 1996 | 4 |
| LU | CA | 1997 | 10 |
| LU | US | 1993 | -100 |
| LU | US | 1996 | 400 |
| LU | US | 1997 | 500 |
| LU | WORLD | 1994 | -100 |

## last_value

*Semantics*

For each sliding window and for each Measure Component, the operator calculates the first value according to the specified order.

*Examples*

ds := **last_value (** ds_bop **) over ( partition by** ref_area , partner **order by** time **)**

| ds_bop | | | |
|---|---|---|---|
| **REF_AREA** | **PARTNER** | **TIME** | **OBS_VALUE** |
| LU | CA | 1993 | 3 |
| LU | CA | 1994 | 4 |
| LU | CA | 1996 | 10 |
| LU | CA | 1997 | 20 |
| LU | US | 1993 | 400 |
| LU | US | 1996 | 500 |
| LU | US | 1997 | 600 |
| LU | WORLD | 1994 | 1000 |

| ds | | | |
|---|---|---|---|
| **REF_AREA** | **PARTNER** | **TIME** | **OBS_VALUE** |
| LU | CA | 1993 | 20 |
| LU | CA | 1994 | 20 |
| LU | CA | 1996 | 20 |
| LU | CA | 1997 | 20 |
| LU | US | 1993 | 600 |
| LU | US | 1996 | 600 |
| LU | US | 1997 | 600 |
| LU | WORLD | 1994 | 1 000 |

In spite of the general syntax for analytic functions, the following ones do not allow the definition of any sliding window, that is, it always coincide with the entire window.

## ntile

*Semantics*

For each Data Point of each window, the operator produces a Data Point where the values of the numeric Measures Components are set to a unique window number. The values of the non-numeric Measure Components are just copied. For each windows a unique number (incrementally generated, starting with 1) is assigned. Note that the order by clause of analytic functions operates within each window; therefore, the windows are not mutually ordered.

*Examples*

ds := **ntile (** ds_bop **) over ( partition by** REF_AREA, partner **order by** time **)**

| ds_bop | | | |
|---|---|---|---|
| **REF_AREA** | **PARTNER** | **TIME** | **OBS_VALUE** |
| LU | CA | 1993 | 3 |
| LU | CA | 1994 | 4 |
| LU | CA | 1996 | 10 |
| LU | CA | 1997 | 20 |
| LU | US | 1993 | 400 |

| | | | |
|---|---|---|---|
| LU | US | 1996 | 500 |
| LU | US | 1997 | 600 |
| LU | WORLD | 1994 | 1000 |

| ds | | | |
|---|---|---|---|
| REF_AREA | PARTNER | TIME | OBS_VALUE |
| LU | CA | 1993 | 1 |
| LU | CA | 1994 | 1 |
| LU | CA | 1996 | 1 |
| LU | CA | 1997 | 1 |
| LU | US | 1993 | 2 |
| LU | US | 1996 | 2 |
| LU | US | 1997 | 2 |
| LU | WORLD | 1994 | 3 |

## percent_rank

*Semantics*

The operator calculates the percent rank of each Data Point with respect to the other Data Points of the same window. For each Data Point and for each numeric Measure Component, the percent rank is calculated as the rank of that Data Point minus one divided by the number of total Data Points in the partition. Data Points with equal values for the ranking criteria receive the same percent rank. The values of each numeric Measure Component are assigned to the respective percent rank. All the values of the non-numeric Measure Components are just copied.

*Examples*

ds := **percent_rank (** ds_bop **) over ( partition by** ref_area, partner **order by** time **)**

| ds_bop | | | |
|---|---|---|---|
| REF_AREA | PARTNER | TIME | OBS_VALUE |
| LU | CA | 1993 | 3 |
| LU | CA | 1994 | 4 |
| LU | CA | 1996 | 10 |
| LU | CA | 1997 | 20 |
| LU | US | 1993 | 400 |
| LU | US | 1996 | 500 |
| LU | US | 1997 | 600 |
| LU | WORLD | 1994 | 1000 |

| ds | | | |
|---|---|---|---|
| REF_AREA | PARTNER | TIME | OBS_VALUE |
| LU | CA | 1993 | 0 |
| LU | CA | 1994 | 0.25 |
| LU | CA | 1996 | 0.5 |
| LU | CA | 1997 | 0.75 |

| | | | |
|---|---|---|---|
| LU | US | 1993 | 0 |
| LU | US | 1996 | 0.33 |
| LU | US | 1997 | 0.67 |
| LU | WORLD | 1994 | 0 |

## rank

*Semantics*

The operator calculates the rank of each Data Point with respect to the other Data Points in the same window.

For each Data Point and for each numeric Measure Component, the rank is calculated as the relative position of the Data Point in the window. The values of each numeric Measure Component is assigned to the respective rank. All the values of the non-numeric Measure Components are just copied.

*Examples*

ds_1 := **rank (** ds_bop **) over ( partition by** ref_area, partner **order by** time **)**

| ds_bop | | | |
|---|---|---|---|
| REF_AREA | PARTNER | TIME | OBS_VALUE |
| LU | CA | 1993 | 3 |
| LU | CA | 1994 | 4 |
| LU | CA | 1996 | 10 |
| LU | CA | 1997 | 20 |
| LU | US | 1993 | 400 |
| LU | US | 1996 | 500 |
| LU | US | 1997 | 600 |
| LU | WORLD | 1994 | 1000 |

| ds_1 | | | |
|---|---|---|---|
| REF_AREA | PARTNER | TIME | OBS_VALUE |
| LU | CA | 1993 | 1 |
| LU | CA | 1994 | 2 |
| LU | CA | 1996 | 3 |
| LU | CA | 1997 | 4 |
| LU | US | 1993 | 1 |
| LU | US | 1996 | 2 |
| LU | US | 1997 | 3 |
| LU | WORLD | 1994 | 1 |

## ratio_to_report

*Semantics*

The operator calculates the percentage amount of the value of each Data Point in the respective window (ratio to report).

For each Data Point and for each numeric Measure Component, the ratio to report is calculated as the percentage amount of the value of Measure Component in the sum of the values for the same Measure Component of the other Data Points in the window. The values of each numeric Measure Component is assigned to the respective ratio to report. All the values of the non-numeric Measure Components are just copied.

*Examples*

ds_1 := **ratio_to_report (** ds_bop **) over ( partition by** REF_AREA, partner**)**

| ds_bop | | | |
|---|---|---|---|
| REF_AREA | PARTNER | TIME | OBS_VALUE |
| LU | CA | 1993 | 3 |
| LU | CA | 1994 | 4 |
| LU | CA | 1996 | 10 |
| LU | CA | 1997 | 20 |
| LU | US | 1993 | 400 |
| LU | US | 1996 | 500 |
| LU | US | 1997 | 600 |
| LU | WORLD | 1994 | 1000 |

| ds_1 | | | |
|---|---|---|---|
| REF_AREA | PARTNER | TIME | OBS_VALUE |
| LU | CA | 1993 | 0.08108 |
| LU | CA | 1994 | 0.10810 |
| LU | CA | 1996 | 0.27027 |
| LU | CA | 1997 | 0.54054 |
| LU | US | 1993 | 0.26667 |
| LU | US | 1996 | 0.33333 |
| LU | US | 1997 | 0.40000 |
| LU | WORLD | 1994 | 0.10000 |

## hierarchy

*Semantics*

VTL foresees the existence of set relations among Code Items in Code List.

Many Enumerated Value Domains have an intrinsic Boolean algebraic structure, in the sense that a Boolean algebra can be defined on the respective Code Items.

In general, a Boolean algebra is an algebraic structure (elements and operators having some properties) that summarizes the properties of set operators (*union*, *intersection*, *complement*) and logical operators (or, and, not).

Elements of a Boolean Value Domain can be combined with the elementary set operators[3]; e.g. the item C is the union of the items D and E; the item K is the complement of S with respect to J (so the elements in J that are not in S), the item A is the intersection of B and C and so on.

Only considering the set union, there are two possible organizations for hierarchical Code Lists: classifications and free hierarchies.

In **classifications**, every element is uniquely classified by a single partition, so that the overall structure is a tree and, consequently, every Code Item can be given a specific level. An example is the usual geographical classification of the world into continents, each partitioned into nations.

[3] Here we refer to elementary set operations and not to any operator of the language.

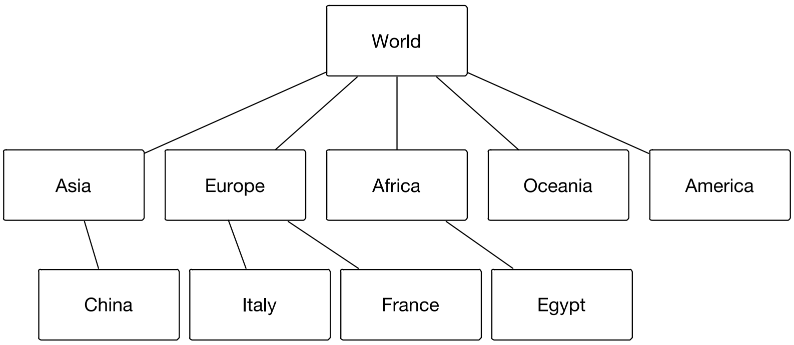



**Figure 1 - example of hierarchical Code List**

In **free hierarchies** each item can be partitioned according to multiple criteria; in turn, every element can be used to compose multiple other elements. The overall structure is not a tree and isolated Items can be present. An example is the hierarchy of European countries, where Belgium, Holland and Luxembourg contribute to the "European Countries" Code Item and to the "Benelux" Code Item.

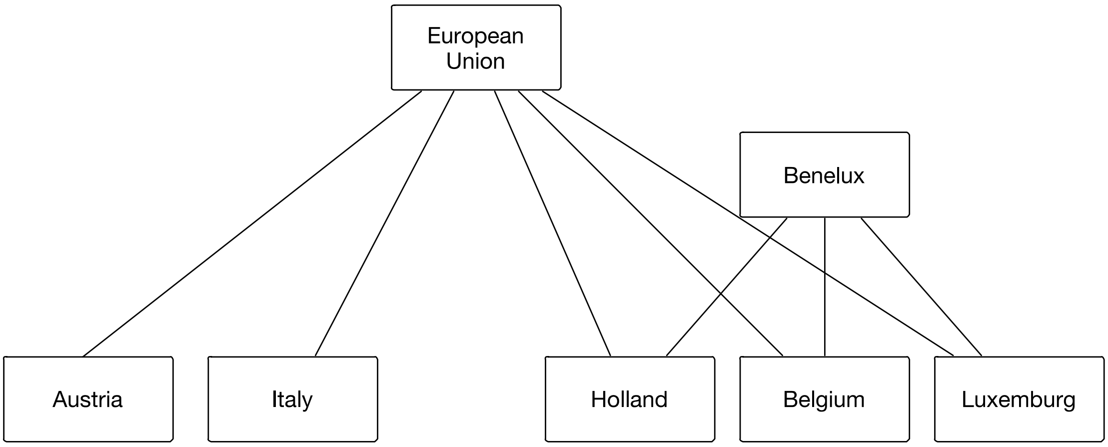



**Figure 2 - Example of free hierarchy**

More sophisticated hierarchical organizations can indeed exist if intersection and complement set operators are also considered. For example the element "Benelux" could also be defined as the complement of the European Union with respect to all the countries except Holland, Belgium and Luxembourg.
Therefore we would have:
BENELUX = EU – (ITALY ∪ AUSTRIA ∪ ... )
In order to support multiple classifications of Code Items in Code Lists and allow for the adoption of all the set operators, we introduce two concepts *hierarchical aggregations* and *mappings* (that will be used in the **hierarchy** operator).
A *hierarchical aggregation* is a set of *mappings*, each transforming a Code Item into another Code Item. All the *mappings* within a *hierarchical aggregation* associate Code Items of the same Code List with Code Items of a single Code List.
Hierarchical aggregations and mappings can be easily expressed in tabular form and referred to in the language (in **hierarchy** operator) by an identifier (*hierarchyName*). However, there is also an inline syntactical form.
Suppose we want to express the hierarchical relationship BENELUX = Belgium ∪ Holland ∪ Luxembourg. There will be a hierarchical aggregation **Benelux_aggr**, with the following mappings:

| HIERARCHY NAME | MAPS FROM | MAPS TO | START DATE | END DATE | SIGN | LEVEL | OUTPUT |
|---|---|---|---|---|---|---|---|
| **Benelux_aggr** | Belgium | Benelux | .... | ... | + | 1 | true |
| **Benelux_aggr** | Holland | Benelux | ... | ... | + | 1 | true |
| **Benelux_aggr** | Luxembourg | Benelux | ... | ... | + | 1 | true |

It maps each component item into the compound one.
Each mapping has a *Sign*. It specifies if the contribution of the MAPS FROM Code Item in the composition is positive (UNION) or negative (COMPLEMENT). Notice that there is not a particular convention to represent intersection, as it can be obtained with an appropriate composition of UNION and COMPLEMENT.
For instance, if we want to define in the element **EuropeWithoutItaly**, a possible definition could be the one complementing Italy, with respect to the entire Europe (i.e. subtracting Italy from Europe), as shown in the following table:

| HIERARCHY NAME | MAPS FROM | MAPS TO | START DATE | END DATE | SIGN | LEVEL | OUTPUT |
|---|---|---|---|---|---|---|---|
| **Eu_no_Italy_aggr** | Europe | EuropeWithoutItaly | …. | … | + | 1 | true |
| **Eu_no_Italy_aggr** | Italy | EuropeWithoutItaly | … | … | - | 1 | true |

Moreover, mappings are divided into levels, in the sense that a complete tree can be embedded into one hierarchy. The OUTPUT property, for each MAPS TO, indicates if the value must be preserved in the output or is only to be used for aggregations at a higher level.
For instance, suppose we want to express the hierarchical relationship in Figure2, there will be a hierarchy **World_aggr**, with the following correspondences:

| HIERARCHY NAME | MAPS FROM | MAPS TO | START DATE | END DATE | SIGN | LEVEL | OUTPUT |
|---|---|---|---|---|---|---|---|
| **World_aggr** | Italy | Europe | …. | … | + | 1 | false |
| **World_aggr** | France | Europe | … | … | + | 1 | false |
| **World_aggr** | Luxembourg | Europe | … | … | + | 1 | false |
| **World_aggr** | … | Europe | | | + | 1 | false |
| **World_aggr** | Asia | World | | | + | 2 | true |
| **World_aggr** | Europe | World | | | + | 2 | true |
| **World_aggr** | America | World | | | + | 2 | true |
| **World_aggr** | Oceania | World | | | + | 2 | true |
| **World_aggr** | Africa | World | | | + | 2 | true |

Output *false* for LEVEL 1, indicates that the aggregations into Europe (and into the other countries) are only functional to LEVEL 2 and the MAPS TO values with *false* will not be in the output.
Summarizing, the set of mappings within a hierarchy has to be interpreted as follows.
For every level, from the lowest to the highest, each MAPS TO Code Item is the set difference between the UNION of all the corresponding MAPS FROM Code Items with positive SIGN and the UNION of all the corresponding MAPS FROM Code Items with the negative SIGN.
*Rules* inherently represent hierarchies in Code Lists, but, at the same time, only refer to Code Items. Thus, can be applied on different *Identifier Components* referring to different Code Lists.

*Syntax*
**hierarchy**
**(** {**dataset=**} ds_1, {**component=**} comp,
[
{**hierarchy_name=**} hierarchyname |
**(** { **( (**{**from=**} maps_from, {**level=**} level, {**sign=**} [**+**|**-**]**)**
{,**(**{**from=**} maps_from, {**level=**} level, {**sign=**} [**+**|**-**]**)**}*)
to {**to=**} maps_to}+ ) as hierarchyname
],

{**isFilter=**} isFilter
{, {**aggregation=**} [**sum**|**prod**] }
**)**

*Parameters*

*ds_1* : Dataset<?,MeasureComponent<Numeric>+>
*comp* : Identifier Component
*hierarchyname* : string
*maps_from, maps_to* : Constant
*level* : integer
*hierarchyname* : string
*isFilter* : boolean

*ds_1* – the Dataset to be aggregated
*comp* – the *IdentifierComponent* to aggregate upon
*hierarchyname* – the name of the hierarchical aggregation of the information model, which can be optionally replaced by an inline specification of the rule
*maps_from* – an input value in an inline aggregation rule
*level* – the level of hierarchy of a correspondence in an aggregation rule
*sign* – the sign of the contribution of a *maps_from* value in an aggregation rule
*maps_to* – an output value in an inline aggregation rule
*isFilter* – if the aggregation must be interpreted as a filter (excluding non matching records)

*Constraints*

- hierarchyname denotes an hierarchical aggregation either externally configured in a table or embedded in the operator with the inline notation (static).
- level > 0 (static).
- All the *MeasureComponents* of ds_1 must be Numeric (static).

*Semantic specification*

It applies a hierarchical aggregation with name *hierarchyname* on an Identifier Component *comp* in a Dataset *ds_1*, aggregating all the Measure Component*s* according to an aggregation function (algebraic sum or product).
*hierarchyname* can be the identifier of an aggregation that is externally configured in a table, with an associated set of mappings, each with a sign and a level.
Alternatively, a hierarchical aggregation can be expressed in an inline fashion, where maps_from constants are mapped into maps_to ones in the specific level and sign.
For the given aggregation, for each level, all the mappings are considered and orderly applied with the following logic.
For each value of the *IdentifierComponent* of all the records of the considered Dataset, if the value is present in *maps_from* for any mapping, it is turned into the respective *maps_to* value; if the value is not present in *maps_from* for any correspondence, the entire record is discarded if isFilter is TRUE, the original value is preserved if *isFilter* is false.
Aggregations are typically hierarchical, in the sense that they map many *maps_from* values into fewer *maps_to* values: often, multiple component Code Items collapse into the same compound Code Item.
Therefore, at this stage, there may be multiple records having the same values for all the *IdentifierComponents*, as the differentiating ones have been aggregated into the same one.
The records having the same value for all the *IdentifierComponents* are aggregated by algebraic sum (**sum**) or product (**prod**) of their *MeasureComponents*. If the aggregation function is omitted, sum is implied.
Sum implies that the *MeasureComponents* have to be algebraically summed, considered with positive or negative sign depending on the Sign of the used mapping.
Prod implies that the *MeasureComponents* have to be multiplied, considered with -1 exponent when the negative Sign is used in the mapping.
Notice that the use of prod as aggregation function is meaningful only when the *IdentifierComponent* is a measure dimension.

**Hierarchies and measure dimensions**

As it is well known, an *IdentifierComponent* in a Dataset can play the role of *measure dimension*, meaning that the Dataset is indeed multi-measure, but represented as mono-measure with a further, measure-qualifying dimension.
*IdentifierComponents* in hierarchy can also be a measure dimension, although in this case the aggregation inherently assumes a different meaning.

In facts, Data Points with all coinciding *IdentifierComponent*s, except for the measure dimension, are nothing but an expression of different measures for the same data point. An aggregation over the *measure dimension* is conceptually an operation involving the measures of the same Data Point (algebraic sum or multiplication).

**Hierarchies as transcodings**

The presented mapping method can be used intuitively to express a transcoding in a synthetic way. In this case, the involved Component value is mapped into another one, which is not hierarchically related with the first, but simply represents the same value expressed in another coding standard.

**Hierarchies as filters**

*isFilter* parameter in *hierarchy* allows choosing whether an input Data Point is to be kept in the output even if there are not mappings having the *maps_from* value corresponding to the Identifier Component of that Data Point.

Indeed, this mechanism lends itself to the construction of reusable filters, independent of the specific Dataset and Identifier Component they are applied on.

Such rules comprise a set of (X→X, Y→, ..., Z→Z) correspondences preserving the values X, Y and Z of the *IdentifierComponents* and filtering out the Data Points having a value for the *IdentifierComponent* different from X, Y or Z.

*Returns*

The aggregated Dataset

*Examples*

1) The expression:

Income_by_country_and_nace_ISO := **hierarchy(**“Income_by_country_and_nace_UN”, GEO, un_to_ISO_aggr, false**)**

or its equivalent inline form:

Income_by_country_and_nace_ISO **:= hierarchy(**“Income_by_country_and_nace_UN”, GEO, ((“ITA”,1,+)) to “IT”, ((“ALB”,1,+)) to AL, ((“BEL”,1,+)) to “BE”) as un_to_ISO_aggr, false**)**

Converts the values of the GEO Identifier Component from the United Nations 3-letter standard into the ISO 2-letter one.

| HIERARCHY NAME | MAPS FROM | MAPS TO | START DATE | END DATE | SIGN | LEVEL | OUTPUT |
|---|---|---|---|---|---|---|---|
| **un_to_ISO_aggr** | ITA | IT | .... | ... | + | 1 | TRUE |
| **un_to_ISO_aggr** | ALB | AL | ... | ... | + | 1 | TRUE |
| **un_to_ISO_aggr** | BEL | BE | ... | ... | + | 1 | TRUE |

2) The expression:

Income_by_continent_and_nace := **hierarchy(**“Income_by_state_and_nace”, GEO, Continent_aggr, false**)**

Takes as input the Dataset of the income, broken down by states and NACE and aggregates to continent level.

The expression:

| HIERARCHY NAME | MAPS FROM | MAPS TO | START DATE | END DATE | SIGN | LEVEL | OUTPUT |
|---|---|---|---|---|---|---|---|
| **Continent_aggr** | Italy | Europe | .... | ... | + | 1 | TRUE |
| **Continent_aggr** | France | Europe | ... | ... | + | 1 | TRUE |
| **Continent_aggr** | Luxembourg | Europe | ... | ... | + | 1 | TRUE |
| **Continent_aggr** | ... | Europe | | | + | 1 | TRUE |
| **Continent_aggr** | China | Asia | | | + | 1 | TRUE |
| **Continent_aggr** | India | Asia | | | + | 1 | TRUE |
| **Continent_aggr** | USA | America | | | + | 1 | TRUE |
| **Continent_aggr** | ... | ... | | | + | 1 | TRUE |

| Income_by_state_and_nace | | |
|---|---|---|
| **GEO** | **NACE** | **VALUE** |
| Italy | IND | 10 |
| Italy | TECH | 20 |
| France | IND | 31 |
| France | TECH | 50 |
| Spain | IND | 30 |
| Spain | TECH | 15 |
| China | IND | 250 |
| China | TECH | 250 |
| India | IND | 30 |
| India | TECH | 100 |
| Luxembourg | IND | 10 |
| Luxembourg | TECH | 12 |

| INCOME_BY_CONTINENT_AND_NACE | | |
|---|---|---|
| **GEO** | **NACE** | **VALUE** |
| Europe | IND | 81 |
| Europe | TECH | 97 |
| Asia | IND | 280 |
| Asia | TECH | 359 |

3) The expression:
Income_by_world_and_nace := **hierarchy(**“Income_by_state_and_nace”, GEO, World_aggr, false**)**
Takes as input the Dataset of the income, broken down by states and NACE and aggregates to the world level. World_aggr is a 2 levels Rule. LEVEL 1 is only a temporary output and its MAPS TO Code Items do not appear in the final result.

| HIERARCHY NAME | MAPS FROM | MAPS TO | START DATE | END DATE | SIGN | LEVEL | OUTPUT |
|---|---|---|---|---|---|---|---|
| **World_aggr** | Italy | Europe | .... | ... | + | 1 | false |
| **World_aggr** | France | Europe | ... | ... | + | 1 | false |
| **World_aggr** | Luxembourg | Europe | ... | ... | + | 1 | false |
| **World_aggr** | ... | Europe | | | + | 1 | false |
| **World_aggr** | ... | Europe | | | + | 1 | false |
| **World_aggr** | China | Asia | | | + | 1 | false |
| **World_aggr** | India | Asia | | | + | 1 | false |
| **World_aggr** | USA | America | | | + | 1 | false |
| **World_aggr** | Asia | World | | | + | 2 | true |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **World_aggr** | Europe | World | | | + | 2 | true |
| **World_aggr** | America | World | | | + | 2 | true |
| **World_aggr** | Oceania | World | | | + | 2 | true |
| **World_aggr** | Africa | World | | | + | 2 | true |

| Income_by_state_and_nace | | |
|---|---|---|
| **GEO** | **NACE** | **VALUE** |
| Italy | IND | 10 |
| Italy | TECH | 20 |
| France | IND | 31 |
| France | TECH | 50 |
| Spain | IND | 30 |
| Spain | TECH | 15 |
| China | IND | 250 |
| China | TECH | 250 |
| India | IND | 30 |
| India | TECH | 100 |
| Luxembourg | IND | 10 |
| Luxembourg | TECH | 12 |

| Income_world | | |
|---|---|---|
| **GEO** | **NACE** | **VALUE** |
| WORLD | IND | 361 |
| WORLD | TECH | 444 |

4) The expression:
Income_world_and_nace_par := **hierarchy(**“Income_by_state_and_nace”, World_aggr_par, false**)**
Or its equivalent inline version:
Income_world_and_nace_par := **hierarchy (** Income_by_state_and_nace, GEO,
(("Italy",1,+),("France",1,+),("Luxembourg",1,+) to "Europe,
(("China",1,+),("India",1,+)) to "Asia",
(("USA",1,+)) to "America,
(("Asia",2,+),("Europe",2,+),("America",2,+),
("Oceania",2,+),("Africa",2,+)) to "World")
as World_aggr_par,
false**)**

Takes as input the Dataset of the income, broken down by states and NACE and aggregates to the World level. Differently from example 2, it preserves the first level in the output.

| HIERARCHY NAME | MAPS FROM | MAPS TO | START DATE | END DATE | SIGN | LEVEL | OUTPUT |
|---|---|---|---|---|---|---|---|
| **World_aggr_par** | Italy | Europe | .... | ... | + | 1 | true |
| **World_aggr_par** | France | Europe | ... | ... | + | 1 | true |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **World_aggr_par** | Luxembourg | Europe | … | … | + | 1 | true |
| **World_aggr_par** | … | Europe | | | + | 1 | true |
| **World_aggr_par** | … | Europe | | | + | 1 | true |
| **World_aggr_par** | China | Asia | | | + | 1 | true |
| **World_aggr_par** | India | Asia | | | + | 1 | true |
| **World_aggr_par** | USA | America | | | + | 1 | true |
| **World_aggr_par** | Asia | World | | | + | 2 | true |
| **World_aggr_par** | Europe | World | | | + | 2 | true |
| **World_aggr_par** | America | World | | | + | 2 | true |
| **World_aggr_par** | Oceania | World | | | + | 2 | true |
| **World_aggr_par** | Africa | World | | | + | 2 | true |

| Income_by_state_and_nace | | |
|---|---|---|
| **GEO** | **NACE** | **VALUE** |
| Italy | IND | 10 |
| Italy | TECH | 20 |
| France | IND | 31 |
| France | TECH | 50 |
| Spain | IND | 30 |
| Spain | TECH | 15 |
| China | IND | 250 |
| China | TECH | 250 |
| India | IND | 30 |
| India | TECH | 100 |
| Luxembourg | IND | 10 |
| Luxembourg | TECH | 12 |

| Income_world_and_nace_par | | |
|---|---|---|
| **GEO** | **NACE** | **VALUE** |
| World | IND | 361 |
| World | TECH | 444 |
| Europe | IND | 81 |
| Europe | TECH | 97 |
| Asia | IND | 280 |
| Asia | TECH | 359 |

5) The expression:
Italian_income_by_nace:= **hierarchy(**“Income_by_state_and_nace”, Italy_filter, true**)**

Takes as input the Dataset of the income, broken down by states and NACE and filters out all the incomes that do not refer to Italy.

| HIERARCHY NAME | MAPS FROM | MAPS TO | START DATE | END DATE | SIGN | LEVEL | OUTPUT |
|---|---|---|---|---|---|---|---|
| **Italy_filter** | Italy | Italy | .... | ... | + | 1 | TRUE |

| Income_by_state_and_nace | | |
|---|---|---|
| GEO | NACE | **VALUE** |
| Italy | IND | 10 |
| Italy | TECH | 20 |
| France | IND | 31 |
| France | TECH | 50 |
| Spain | IND | 30 |
| Spain | TECH | 15 |
| China | IND | 250 |
| China | TECH | 250 |
| India | IND | 30 |
| India | TECH | 100 |
| Luxembourg | IND | 10 |
| Luxembourg | TECH | 12 |

| Italian_income_by_nace | | |
|---|---|---|
| GEO | NACE | **VALUE** |
| Italy | IND | 10 |
| Italy | TECH | 20 |

6) The expression:
income_prod_2:= **hierarchy(**"Income_by_state_and_nace_mm", GEO, mult_rule, false, prod**) .**
Takes as input the Dataset of the income, broken down by states and NACE in a measure dimension form and aggregates by calculating INC2 / INC1 into INC.

| HIERARCHY NAME | MAPS FROM | MAPS TO | START DATE | END DATE | SIGN | LEVEL | OUTPUT |
|---|---|---|---|---|---|---|---|
| **Mult_measure** | INC1 | INC | .... | ... | - | 1 | TRUE |
| **Mult_measure** | INC2 | INC | .... | ... | + | 1 | TRUE |

| Income_by_state_and_nace | | | |
|---|---|---|---|
| GEO | NACE | MEASURE | **VALUE** |
| Italy | IND | INC1 | 10 |
| Italy | IND | INC2 | 20 |
| Italy | TECH | INC1 | 20 |

| | | | |
|---|---|---|---|
| Italy | TECH | INC2 | 40 |
| France | IND | INC1 | 31 |
| France | IND | INC2 | 61 |
| France | TECH | INC1 | 50 |
| France | TECH | INC2 | 100 |
| China | IND | INC1 | 250 |
| China | IND | INC2 | 500 |
| China | TECH | INC1 | 250 |
| China | TECH | INC1 | 500 |
| India | IND | INC1 | 30 |
| India | IND | INC2 | 60 |
| India | TECH | INC1 | 100 |
| India | TECH | INC2 | 200 |

| Income_prod2 | | | |
|---|---|---|---|
| **GEO** | **NACE** | **MEASURE** | **VALUE** |
| Italy | IND | INC | 2 |
| Italy | TECH | INC | 2 |
| France | IND | INC | 2 |
| France | TECH | INC | 2 |
| China | IND | INC | 2 |
| China | TECH | INC | 2 |
| India | IND | INC | 2 |
| India | TECH | INC2 | 2 |

# VTL-ML - Data validation functions

## check

### check (with datapoint rulesets)

*Semantics*

This **check** operator applies one or more datapoint Ruleset on a Dataset.

*Syntax*

**check** (
ds ,
*dpr+*
{ , **<u>not valid</u>** | **valid** | **all** }
{ , **<u>condition</u>** | **measures** }
)

*Parameters*

*ds* : dataset-type

- *ds* is the Dataset to check
- *dpr* is a data point Ruleset
- **valid** returns the valid data points of ds according to *dpr*
- **not valid** returns the not valid data points of ds according to *dpr* (default)
- **all** returns all data points of *ds*, independently of whether a specific rule of a Ruleset is respected or not
- **condition** returns a Boolean Measure named **CONDITION** with true values for the Data Points that satisfy the a specific rule of a Ruleset and false otherwise
- **measures** returns the original Measures and attributes of *ds* (default). The parameter mMasures cannot be used in combination with **all**.

*Constraints*

*ds* has exactly the variables that are defined in the signature of *dpr*.

*Returns*

It Returns a Dataset having the same identifiers as the input Dataset plus the RULE_ID identifier. The values of RULE_ID are the concatenation of the name of the Data Point ruleset applied (which is meaningful, since the operator can apply multiple rules to the same Dataset) and the name of the rule within the Ruleset. The Measures returned depend on the specified option:

- With the ***condition*** option: returns a Dataset having all Identifier Components of *ds*, the Identifier Component RULE_ID, the Measure **CONDITION** with a Boolean value holding the outcome of the validation and the attributes ERRORMESSAGE and ERRORLEVEL. If ERRORMESSAGE and ERRORLEVEL are not specified in the datapoint (horizontal) rule the values of the related attributes will be NULL for all the data points .
- With the ***measures*** option: returns a Dataset having all Identifier Components of *ds*, the Identifier RULE_ID, all the original Measures of *ds* and the attributes ERRORMESSAGE and ERRORLEVEL. If ERRORMESSAGE and ERRORLEVEL are not specified in the datapoint ruleset the values of the related attributes will be NULL for all the data points.

The attributes of the Dataset will be:

- ERRORMESSAGE, containing the error message specified in the rule
- ERRORLEVEL, containing the error level (severity) specified in the rule

If ERRORMESSAGE and ERRORLEVEL are not specified in the datapoint (horizontal) rule the values of the related attributes will be NULL for all the data points .
If **not valid** is specified then **check** returns the data points of *ds* that do not satisfy at least a rule of *dpr*. If a data point in ds does not satisfy several data rules then several data points are returned, one for each rule that is not satisfied, with the associated rule_id, error message and error level.
If **valid** is specified then **check** returns the data points of *ds* that do satisfy all rules of *dpr*.
If **all** is specified then **check** returns all data points of *ds*. This option is normally used in combination with **condition.**
See also the examples under **define datapoint ruleset**.

*Examples*

| ds_labour | | | |
|---|---|---|---|
| **TIME** | **PERSON_ID** | **AGE** | **EDU** |
| 2011 | 1 | 15 | 5 |
| 2011 | 2 | 20 | 3 |
| 2011 | 3 | 17 | 6 |
| 2011 | 4 | 19 | 4 |
| 2011 | 5 | 32 | 6 |
| 2011 | 6 | 17 | 14 |
| 2011 | 7 | 25 | 14 |
| 2011 | 8 | 18 | 10 |
| 2011 | 9 | 15 | 3 |
| 2011 | 10 | 40 | 5 |

Where the variable edu is coded using the following classification:

| edu | descr |
|---|---|
| 1 | No title |
| 2 | Elementary License / Certificate of final evaluation |
| 3 | Middle School (or professional training)/ Diploma in Education secondary level |
| 4 | Professional qualifications Diploma of secondary school 2-3 years which does not allow enrollment at the University |
| 5 | High School Diploma / Secondary Education degree higher than 4-5 years which allows enrollment at the University |
| 6 | Academy Diploma (Fine Arts, Dramatic Arts National, National Dance), Higher Institute of Artistic Industries, State Conservatory of Music |
| 7 | University degree of two / three years, direct school for special purposes, school equivalent education |
| 8 | bachelor's degree (three years) |
| 9 | Specialist / Master's degree (two years) |
| 10 | 4-6 years Degree: Bachelor's degree from the old system or graduate specialization / teaching single-cycle |
| 13 | VET Certificate of professional qualification (operator) / Professional diploma IFP technical |

| | |
|---|---|
| | (Three-year pathways / four-year education and training) |
| 14 | higher technical specialization certificate (HTE) |
| 15 | Higher Technical Diploma (ITS) |

```
define datapoint ruleset  dpr_ labour ( AGE as a) (
        rule_1 when a between 14 and 17 then edu <> 5 ;
        rule_2 when a between 16 and 19 then edu <> 6 ;
        rule_3 when a between 17 and 20 then edu <> 7 and edu <> 8 ;
        rule_4 when a between 18 and 21 then edu <> 10 ;
        rule_5 when a between 14 and 16 then edu <> 13 ;
        rule_6 when a between 16 and 18 then edu <> 14 ;
        rule_7 when a between 17 and 20 then edu <> 15 ;
) ;
```

ds_validation_report := check ( ds_labour, dpr_labour, **not valid**, **condition** )

| ds_validation_report | | | | | | | |
|---|---|---|---|---|---|---|---|
| **TIME** | **PERSON_ID** | **RULE_ID** | **AGE** | **EDU** | **CONDITION** | **ERRORMESSAGE** | **ERRORLEVEL** |
| 2011 | 1 | hr_labour_rule_1 | 15 | 5 | FALSE | | |
| 2011 | 3 | hr_labour_rule_2 | 17 | 6 | FALSE | | |
| 2011 | 6 | hr_labour_rule_6 | 17 | 14 | FALSE | | |
| 2011 | 8 | hr_labour_rule_4 | 18 | 10 | FALSE | | |

## check (with hierarchical rulesets)

*Semantics*

This **check** operator applies one or more hierarchical (vertical) ruleset on a Dataset.

*Syntax*

**check** (
*ds,*
*hr+*
{ , **threshold (** *threshold* **)** }
{ , **<u>not valid</u>** | **valid** | **all** }
{ , **<u>measures</u>** | **condition** }
)

*Parameters*

*ds* : dataset-type
*threshold* : numeric-constant

- *ds* is the Dataset to check
- *hr* is a code item compatibility ruleset
- *threshold* is the threshold (tolerance value) to be applied as the upper limit of the difference between the left and right side of the rules. In the simplest case *threshold* is a numeric constant. A more sophisticated form exists where *threshold* is an expression involving the following predefined values:
    - left_side  the value of the left-hand side of the rule

- right_side the value of the right-hand side of the rule (the value computed by VTL)
- nr_items the number of items in the right-hand side of the rule

Examples of possible threshold expressions:
threshold ( 3 )
threshold ( abs ( left_side - right_side ) > 3 ) equivalent to the above */
threshold ( abs ( left_side / right_side ) > abs ( 50 * left_side ) ) can differ up to 50 % of left side*/
threshold ( abs ( left_side - right_side ) > 0.5 * nr_items ) can differ up to 0.5 for each item % */

- **valid** returns the valid data points of ds according to vr
- **not valid** return the non valid data points of ds according to *hr* (default)
- **all** returns all data points of *ds*, independently of whether a specific rule of a Ruleset is respected or not
- **condition** returns a Boolean Measure Boolean attribute "condition" named **CONDITION** with true values for the Data Points that satisfy a specific rule of a ruleset and false otherwise
- **measures** returns the original Measures and attributes of *ds* (default). The parameter **measures** cannot be used in combination with **all.**

*Constraints*

- *ds* has the variables specified in the code item compatibility ruleset *hr*.
- The ruleset must be explicitly defined for validation purposes.

*Returns*

It returns a Dataset having all Identifier Components of *ds* plus the Identifier RULE_ID, which allows to distinguish between validation rules in the various rulesets when ambiguities arise. The values of RULE_ID are built as the concatenation of the ruleset identifier and the rule identifier.
The Measures of the Dataset returned depend on the specified option:

- With the **condition** option: it returns a Dataset having the Measure **CONDITION** (true/false)
- With the **measures** option: it returns a Dataset having all the Measures of *ds*

Additional Measures are:

- value_CONDITION: containing the value computed by executing the rule
- rule: containing the returning text related to the rule

The attributes of the Dataset will be:

- errorcode, containing the error message specified in the rule
- errorlevel, containing the error level (severity) specified in the rule

If errormessage and errorlevel are not specified in the ruleset, then the values of the related attributes will be NULL for all the data points .

*Examples*

See to the examples for hierarchical (vertical) rulesets.

## check ( single rule)

*Semantics*

The **check** operator takes as input a Boolean VTL expression and uses it as the indication of a validation.

*Syntax*

**check** (
*ds*
{ , **threshold** (*threshold*) }
{ , <u>**not valid**</u> | **valid** | **all** }

{ , **<u>measures</u>** | **condition** }
{ , **imbalance** ( *imbalance* ) }
{ , **errorcode** ( *errorcode*) }
{ , **errorlevel** ( *errorlevel* ) }
)

*Parameters*
ds : dataset {identifier <IDENT> as scalar-type; }+ {measure <IDENT> as numeric; }+
{measure <IDENT> as boolean-type; }* {attribute <IDENT> as scalar-type; }*
*threshold* : scalar-constant
*imbalance* : numeric-constant
*errormessage* : string-constant
*errorlevel* : integer-constant

- *ds* is the Boolean VTL expressions, hence yielding a Boolean Dataset, that represents the validation.
- *threshold* is a tolerance number. It requires the presence of an imbalance. If this latter value is below the *threshold*, then the data point is considered valid, thus the Boolean Measure **CONDITION** is true; false in all the other cases.
- **valid** returns only the valid data points
- **not valid** returns only the non valid data points
- **all** returns all data points, independently of their validity
- **condition** returns a Boolean Measure "**CONDITION**" with values true for the data points that satisfy the ruleset and false otherwise
- **measures** returns a Boolean Measure "**CONDITION**" with values true for the data points that satisfy the ruleset and false otherwise (default)
- *imbalance* is the imbalance to be computed. Imbalance has a number datatype. If not specified in the check, it will be not in the output.
- *errorcode* is the error code to be produced when the validation fails.
- *errorlevel* is the error level (severity) of the validation rule. Errorlevel has a string datatype. If not specified in the check, it will be not in the output.

*Constraints*
- The input Dataset must have all Boolean Measure Components.
- When a threshold is specified, the imbalance must be specified as well.

*Returns*
It returns a Dataset with all the Identifier Components of ds .
The measures of the Dataset returned, depend on the option specified:
- With the **condition** option: returns the Measure **CONDITION** (true/false) .
- With the **measures** option: returns all the Measures of *ds* .

Additional measures are:
- **CONDITION**, when the the Boolean value computed by executing the rule (true/false) (depending on the optional parameter (condition or Measures):
- imbalance, imbalance to be computed

The attributes of the Dataset will be:
- errorcode, containing the error code specified in the rule
- errorlevel , containing the error level (severity) specified in the rule

*Semantic specification*

It takes as input a Boolean VTL expression and uses it as the indication of a validation. It returns an output Dataset that specifies the outcome of the validation. It can report in the output Dataset what are the violations, what are the original data, what is the imbalance between the expected values and the actual ones also applying thresholds. It can also link the failed validations to specific error codes and error levels for further processing.

*Examples*

| ds_bop | | | | |
|---|---|---|---|---|
| TIME | REF_AREA | PARTNER | FLOW | OBS_VALUE |
| 2010 | IT | US | NET | 10 |
| 2011 | IT | US | NET | 20 |
| 2012 | IT | US | NET | 50 |
| 2013 | IT | US | NET | 40 |
| 2014 | IT | US | NET | 50 |
| 2015 | IT | US | NET | 60 |
| 2010 | DE | US | NET | 25 |
| 2011 | DE | US | NET | 35 |
| 2012 | DE | US | NET | 45 |
| 2013 | DE | US | NET | 55 |
| 2014 | DE | US | NET | 65 |
| 2015 | DE | US | NET | 75 |

Check that the difference between each value and the average of that value, its preceding value and following value is lower than 10 (in absolute value).

```
ds_moving_average := avg ( ds_bop  )  over (
        partition by ref_area, partner
        order by time
        rows between 1 preceding and 1 following ) ;
ds_outliers := check ( abs ( ds_bop.obs_value – ds_moving_average.obs_value ) <= 10 ) ;
```

| ds_outliers | | | | | | | |
|---|---|---|---|---|---|---|---|
| TIME | REF_AREA | PARTNER | FLOW | OBS_VALUE | IMBALANCE | ERRORMESSAGE | ERRORLEVEL |
| 2012 | IT | US | NET | 50 | | | |

# check value domain subset

*Semantics*

The **check_value_domain_subset** operator checks if the values of the specified Components owned by *the* Dataset are part of the restricted domain of the ValueDomain.

*Syntax*

**check_value_domain_subset (** *ds*, [ *components* | { *compList* **(**{*compIndent*}+**)**, *valueDomain* }], *vds***);**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*

*vds* : valueDomainSubset-ref
*components* : list <component-ref>
*compList* : list<list<component-ref>>
*compIndentm*: list <component-ref>
*valueDomainm*: list <dimension-ref>

- *ds* – is the starting Dataset.
- *components* - is the List containing the Components owned by ds to be validated.
- *compList* – is the list containing the Components (divided in lists *compIndent*) that must be checked according to the restrictions defined in the ValueDomainSubset *vds* passed as input.
- *valueDomain* – is the list containing the dimension of the ValueDomainSubset that are used to validate the Components referred in *compList* and *compIndent*
- *vds* – is the ValueDomainSubset containing the restrictions to be verified in *ds*.

*Constraints*
- if only components are defined then *vds* must be mono-dimensional.
- if both *compList* and *compIndent* are defined they must be of the same dimension.

*Returns*
A Dataset with all the Identifier, Measure and Attribute Components of the input one enriched by a Boolean Measure Component for each Component specified in *components* or *compList*, that contains the result of the check, against the ValueDomainSubset restrictions, for the respective values.

*Semantic specification*
The operator checks if the values of the specified Components owned by *ds* are part of the restricted domain of the ValueDomain in *vds*, returning a Dataset with the same structure of the input one and a Boolean Measure Component for each Component specified in the signature (the name of the new Component is "COMPONENT_NAME" + "**_CONDITION**"). For each Datapoint the new Boolean Measure Component assumes the value TRUE if the value of the respective Component is part of the restricted domain of the respective ValueDomain in *vds,* FALSE otherwise.
The operator can work in two mode: mono-dimensional and multi-dimensional mode.
In the mono-dimensional version (only *components* defined) it takes as input a Dataset, a List of Components and a mono-dimensional ValueDomainSubset. It evaluates if all the values inside the specified Components of *ds* are part of the restricted domain defined by the mono-dimensional ValueDomainSubset.
In the multi-dimensional version (both *compList* and *valueDomains* defined) it takes as input a Dataset, two Lists and a multi-dimensional ValueDomainSubset. The first list is a List of Lists containing names of components owned by *ds*, the second List contains reference to the ValueDomain owned by *vds*. The Components specified in the first element of *compList* will be checked against the ValueDomain specified in the first element of *valueDomains*, and so on (it follows that the two Lists must have the same size and order of the elements matters).

*Examples*
1)

| ds_1 | | | | |
|---|---|---|---|---|
| **TIME** | **REF_AREA** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | CA | 20 | D |
| 2010 | BG | CA | 1 | P |
| 2010 | RO | CA | 1 | P |
| 2010 | EU27 | CA | 23 | P |

l_1 = list<components-ref> (REF_AREA)

ds_1 := **check_value_domain_subset (**ds_1, l_1, vds_1**)** vds_1 is a mono-dimensional enumerated ValueDomainSubset, the CodeList referenced by its ValueDomain contains the values: [“EU25”,”EU27”,”EU28”]

| ds_2 | | | | | |
|---|---|---|---|---|---|
| **TIME** | **REF_AREA** | **REF_AREA_CONDITION** | **PARTNER** | **OBS_VALUE** | **OBS_STATUS** |
| 2010 | EU25 | TRUE | CA | 20 | D |
| 2010 | BG | FALSE | CA | 1 | P |
| 2010 | RO | FALSE | CA | 1 | P |
| 2010 | EU27 | TRUE | CA | 23 | P |

2)
compList := list<list<component-ref>>(list<component-ref>(REF_AREA), list<component-ref>(OBS_VALUE))
ds_2 := **check_dataset_values (**ds_1,  compList, list<valueDomain-ref>(D1, D2), vds_1**)**

vds_1 is a multi-dimensional ValueDomainSubset with two dimensions D1 and D2. D1 take its domain from the values of a CodeList defined as [“EU25”,”EU27”,”EU28”], D2 is a numeric restricted domain that allows only positive integers.

Returned Dataset:

| ds_2 | | | | | | |
|---|---|---|---|---|---|---|
| **TIME** | **REF_AREA** | **REF_AREA_CONDITION** | **PARTNER** | **OBS_VALUE** | **OBS_VALUE_CONDITION** | **OBS_STATUS** |
| 2010 | EU25 | TRUE | CA | 20 | TRUE | D |
| 2010 | BG | FALSE | CA | 1 | TRUE | P |
| 2010 | RO | TRUE | CA | 1 | TRUE | P |
| 2010 | EU27 | FALSE | CA | 23 | TRUE | P |

# VTL-ML - Time series functions

## fill_time_series

*Semantics*

The operator **fill_time_series** replaces each missing data point in the input Dataset (from the lowest to the highest time period found in the Dataset) with a data point having the values of Measures and attributes set to null.

*Syntax*

**fill_time_series ( ***ds, freq,* { , *timePeriodName* { , *timeFormat* } **)**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+ {identifier <IDENT> as date}
{measure <IDENT> as numeric}+
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*

*ds* – is the input Dataset whose missing data points in the series will be filled in.

*Constraints*

The Dataset *ds* must have the specified the Identifier Component *timePeriodName* or the default "time".

*Returns*

A Dataset having the same Identifier, Measure and Attribute Components as the input one. The missing data points in each series will be filled in.

*Semantic specification*

fill_time_series allows to fill in all series of *ds* (no need to process the series one by one).
The time format can be specified as described in the table under "Time aggregate functions".

*Examples*

ds_bop

| TIME | REF_AREA | PARTNER | FLOW | OBS_VALUE |
|---|---|---|---|---|
| 2010 | IT | US | NET | 10 |
| 2012 | IT | US | NET | 50 |
| 2010 | DE | US | NET | 25 |

ds_fill_ts := fill_time_series ( ds_bop, "A", time )

ds_fill_ts

| TIME | REF_AREA | PARTNER | FLOW | OBS_VALUE |
|---|---|---|---|---|
| 2010 | IT | US | NET | 10 |
| 2011 | IT | US | NET | null |
| 2012 | IT | US | NET | 50 |
| 2010 | DE | US | NET | 25 |
| 2011 | DE | US | NET | null |
| 2012 | DE | US | NET | null |

Note: the Dataset contains data from 2010 to 2012 therefore 1 data point is inserted for the series (IT, US, NET) and 2 data points are inserted for the series (DE, US, NET).

# flow_to_stock

*Semantics*

The operator **flow_to_stock** consists in the transformation from a flow interpretation of a Dataset (with one single date Identifier Component), where the numeric Measures represent relative modifications of the stock level (flow), to the corresponding stock interpretation of it, where the numeric Measures represent stock levels Measured at a specific time (stock).

*Syntax*

**flow_to_stock ( *ds* )**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+ {identifier <IDENT> as date}
{measure <IDENT> as numeric}+
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*

*ds* – the input Dataset containing time series.

*Constraints*

- *ds* must be a Dataset that contains only one date Identifier Component (as in the syntax).
- *ds* must be regular, that is, once the Data Points have been ordered by the only date Identifier, for each pair of consecutive Data Points, the distance in time between the respective date values must be constant (and typically one year, one quarter, one day and so on).

*Returns*

A Dataset having the same Identifier, Measure and Attribute Components as the input one. The values of the numeric Measure Components are computed as the stock interpretation of the respective input values, which are considered according to a stock interpretation.

*Semantic specification*

The operator takes as input a Dataset *ds* and returns another one with the same Identifier, Measure and Attribute Components as the input one. We say that two Data Points *dp1* and *dp2* of *ds* are consecutive if they have the same values for all the Identifier Components but the one with date data type and, once all the Data Points with the same values for all the Identifier Components have been ordered by date Component, they are adjacent.
The Data Points of the output are calculated as follows. Data Points in *ds* are partitioned in blocks having the same values for all the Identifier Components but the date one. For each block, the first Data Point is copied into the output. Then, for each pair of consecutive Data Points *dp1* and *dp2* (that is, *dp2* follows *dp1*) of *ds*, a new data Point appears in the output. The value of all the Identifier Components, non-numeric Measure Components and Attribute Components of the output Dataset are copied from *dp2*. The value of each numeric Measure Component is calculated as the sum of the value in the output of the previously copied Data Point and the value of the Measure Component of *dp2*. Note that the operator actually performs the cumulative sum and no Data Points are neglected.

*Examples*

| ts_1 | |
|---|---|
| **DATE** | **VALUE** |
| 1939-01-01 | 4400.0 |
| 1939-02-01 | 0.0 |
| 1939-03-01 | 6200.0 |
| 1939-04-01 | -38000.0 |

ts_2 := **flow_to_stock (** ts_1 **)**

| ts_2 | |
|---|---|
| **DATE** | **VALUE** |
| 1939-01-01 | 4400.0 |
| 1939-02-01 | 4400.0 |
| 1939-03-01 | 10600.0 |
| 1939-04-01 | 6800.0 |

# stock_to_flow

*Semantics*

The operator **stock_to_flow** consists in the transformation from a stock interpretation of a Dataset (with one single date Identifier Component), where the numeric Measures represent stock levels measured at a specific time (stock), to the corresponding flow interpretation of it, where the numeric Measures represent relative modifications of the stock level (flow).

*Syntax*

**stock_to_flow ( *ds* )**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+ {identifier <IDENT> as date}
{measure <IDENT> as numeric}+
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*

*ds* – the input Dataset.

*Constraints*

- *ds* must be a Dataset that contains only one date Identifier Component (as in the syntax).
- *ds* must be regular, that is, once the Data Points have been ordered by the only date Identifier, for each pair of consecutive Data Points, the distance in time between the respective date values must be constant (and typically one year, one quarter, one day and so on).

*Returns*

A Dataset having the same Identifier, Measure and Attribute Components as the input one. The values of the numeric Measure Components are computed as the flow interpretation of the respective input values, which are considered according to a stock interpretation.

*Semantic specification*

The operator takes as input a Dataset *ds* and returns another one with the same Identifier, Measure and Attribute Components as the input one. We say that two Data Points *dp1* and *dp2* of *ds* are consecutive if they have the same values for all the Identifier Components but the one with date data type and, once all the Data Points with the same values for all the Identifier Components have been ordered by date Component, they are adjacent.
The Data Points of the output are calculated as follows. Data Points in *ds* are partitioned in blocks having the same values for all the Identifier Components but the date one. For each block, the first Data Point is copied into the output. Then, for each pair of consecutive Data Points *dp1* and *dp2* (that is, *dp2* follows *dp1*) of *ds*, a new data Point appears in the output. The value of all the Identifier Components, non-numeric Measure Components and Attribute Components of the output Dataset are copied from *dp2*. The value of each numeric Measure Component is calculated as the difference of the respective numeric Measure.

*Examples*

| **ts_1** | |
|---|---|
| **DATE** | **VALUE** |
| 1939-01-01 | 4400.0 |
| 1939-02-01 | 4400.0 |
| 1939-03-01 | 10600.0 |
| 1939-04-01 | 6800.0 |

ts_2 := **stock_to_flow(** ts_1 **)**

| **ts_2** | |
|---|---|
| **DATE** | **VALUE** |
| 1939-01-01 | 4400.0 |
| 1939-02-01 | 0 |
| 1939-03-01 | 6200.0 |
| 1939-04-01 | -3800.0 |

## timeshift

*Semantics*

The operator **timeshift** returns the input Dataset with its time component shifted by the amount of time specified as parameter.

*Syntax*

**timeshift (** *ds*, *timeId*, **unit = [A|M|Q|D],** *lag* **)**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+
{[identifier|measure] <IDENT> as date}
{measure <IDENT> as scalar-type}* {attribute <IDENT> as scalar-type}*
*timeId* : component-ref
*lag* : integer

- *ds* – the input Dataset containing time series.
- *timeId* – is the reference to a valid Component representing the time of the time series of ds, that is the Component on which the shift operation must be performed.
- *unit* – represents the unit of time to be shifted. The possibilities are: Y=year, M=Month, Q=Quarter, D=Day

*Constraints*

*timeId* must refer to a Component of ds, whose type is date (as in the syntax).

*Returns*

A Dataset having the same Identifier, Measure and Attribute Components as the input one, with each value of the *timeId* Component modified of *lag* units.

*Semantic specification*

The operator takes as input a Dataset, a valid date Component of *ds* (*timeId*) and the specification of the **unit** and amount of time (*lag*) to shift. It returns a Dataset with the same structure as the input one. For each Data Point in *ds*, the result contains the same Data Points (so with the same values for all the Identifier, Measure and Attribute Components), except for the values of the Identifier Component *timeId*, which are modified by summing the relative amount **unit** x *lag* (note that *lag* may also be negative).

*Examples*

| ts_1 | |
|---|---|
| **DATE** | **VALUE** |
| 1939-01-01 | 4400.0 |
| 1939-02-01 | 4400.0 |
| 1939-03-01 | 10600.0 |
| 1939-04-01 | 6800.0 |

ts_2 **:= timeshift(** ts_1, A, 1 **)**
equivalent forms:
ts_2 := **timeshift(** ts_1, M,12 **)**
ts_2 := **timeshift(** ts_1, Q, 3 **)**

| ts_2 | |
|---|---|
| **DATE** | **VALUE** |
| 1940-01-01 | 4400.0 |
| 1940-02-01 | 4400.0 |
| 1940-03-01 | 10600.0 |
| 1940-04-01 | 6800.0 |

# VTL-ML - Conditional operators

## if-then-else

*Semantics*

The operator **if-then-else** returns one of two possible dataset or values depending on the input conditions.

*Syntax*

**if** ds_*cond_1* **then** *ds_1* { **elseif** ds_*cond_2* **then** *ds_2* }* **else** *ds_3*

*Parameters*

ds_cond_1, ds_cond_2
: [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as boolean}|boolean]
ds_1, ds_2, ds_3
: [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}+|constant]

- *ds_cond_1 – is the first Boolean condition*
- *ds_1 – can be:*
  - *a Dataset from which the Data Points are retrieved anytime that ds_cond_1 evaluate true.*
  - *a constant constant returned if ds_cond_1 evaluated true.*
- *ds_cond_2 – is an optional Boolean condition*
- *ds_2 – can be:*
  - *a Dataset from which the Data Points are retrieved anytime that ds_cond_2 evaluate true*
  - *a constant constant returned if ds_cond_2 evaluated true.*
- *ds_3 – can be:*
  - *a Dataset from which the Data Points are retrieved anytime that ds_cond_2 not evaluate true.*
  - *a constant constant returned if ds_cond_2 not evaluated true*

*Constraints*

- If *ds_1*, *ds_2* and *ds_3* are constant values, they must have the same type.
- If *ds_cond_1*, *ds_cond_2*, *ds_1*, *ds_2* and *ds_3* are Datasets, they must have the same Identifier Components, in name and type.
- If *ds_cond_1* and *ds_cond_2* are Datasets, they must have only one boolean Measure Component (as expressed in the syntax).
- If *ds_1*,*ds_2* and *ds_3* are Datasets, they must have the same Measure Components.

*Returns*

If the input parameters are Boolean scalars then the operator returns the constant of the first evaluated true condition. If no condition evaluates true then *c_3* is returned.
If the input parameters are Datasets then the operator returns a Dataset having all the Identifier and Measure Components of the input ones, composed by Data Points retrieved in the input Datasets when the relative condition on the relative boolean Measure Component evaluate true.

*Semantic specification*

If the input parameters are Boolean scalars then the operator takes as input a series of Boolean condition with the respective values to return in case of positive validation, it returns the constant of the first evaluated true condition or *c_3* if not a condition evaluate true.
If the input parameters are Datasets then the operator takes as input a number of condition Datasets *ds_cond_1*, having exactly one boolean Measure Component and, for each of them a Dataset *ds* to return in case of positive evaluation (**then**) of the condition. Besides it takes in input a default case (**else**) Dataset to be returned if all the previous conditions evaluate to False. Starting from *ds_cond_1*, for each Data Point, if the Measure Component is True, it looks up in the corresponding Datasets (*ds_1*) all the Data Point for the corresponding values of the Identifier Components and returns them in the output Dataset. If the Measure Component is False, it looks up in the following **elseif** Dataset (*ds_cond_2*) for the corresponding values of the Identifier Components. If no Data Point is found, the elaboration skips to the next Data Point of *ds_cond_*1. If any Data Points are found and *ds_cond_2* is True for them, the corresponding **then** Dataset (*ds_2*) is returned; otherwise, the evaluation continues likewise, until the **else** part is reached (in case every previous conditional Datasets evaluate to False)

and, if any matching Data Points are found in *ds_3* they are returned. Then the elaboration is repeated for all the Data Points in *ds_cond_1*.

*Examples*

On scalar

1) Expressions evaluating to Component types are typically used to calculate Measure Components or evaluate filters.

Some examples follow:

K1 + K2 < K3
K1 – K2 > 5.5
K2 + round(K2, 3)
K1 > 3 and k1 < 5
**if** k1>4 **then** K2 **else** K3 + 3
K1 in (1,2,3,4) and K3 not in (‘a’,’b’,’c’)

On Datasets

2) ds_1 := **if** (population.SEX=”F”).CONDITION
**then** unemp_rates_1
**else** unemp_rates_2

| population | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | M | 5451780 |
| 2012 | Belgium | Total | F | 5643070 |
| 2012 | Greece | Total | M | 5449803 |
| 2012 | Greece | Total | F | 5673231 |
| 2012 | Spain | Total | M | 23099012 |
| 2012 | Spain | Total | F | 23719207 |
| 2012 | France | Total | M | 31616281 |
| 2012 | France | Total | F | 33671580 |
| 2012 | Italy | Total | M | 28726599 |
| 2012 | Italy | Total | F | 30667608 |
| 2012 | Austria | Total | M | NULL |
| 2012 | Austria | Total | F | NULL |

| unemp_rates_1 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **RATE** |
| 2012 | Spain | Total | F | 25.8 |
| 2012 | France | Total | F | NULL |
| 2012 | Italy | Total | F | 20.9 |
| 2012 | Austria | Total | M | 6.3 |

| unemp_rates_2 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **RATE** |
| 2012 | Belgium | Total | M | 0.12 |
| 2012 | Greece | Total | M | 22.5 |
| 2012 | Spain | Total | M | 23.7 |
| 2012 | Austria | Total | F | NULL |

| ds_1 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **RATE** |
| 2012 | Belgium | Total | M | 0.12 |
| 2012 | Greece | Total | M | 22.5 |
| 2012 | Spain | Total | M | 23.7 |
| 2012 | Spain | Total | F | 25.8 |
| 2012 | France | Total | F | NULL |
| 2012 | Italy | Total | F | 20.9 |

population.SEX allows to consider SEX into the only Measure Component, which is compared with “F” by the operator “=”. Correctly, it acts on the only Measure Component as POPULATION is temporarily not considered. The comparison returns CONDITION as the only boolean Measure Component
Thus, as the if operators requires a Dataset with a single boolean Measure Componen, the membership operator "." is applied again in order to isolate SEX_ CONDITION as the only boolean Measure Component

# nvl

*Semantics*
The operator **nvl** replaces null values with a value given as a parameter.

*Syntax*
**nvl (***ds*, rep_*value* **)**

*Parameters*
*ds* : [dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}+
{attribute <IDENT> as scalar-type}*|scalar-type]
*rep_value :* scalar-type

- *ds* can be an scalar-type value or an input Dataset
- *rep_value* is the value that replace the values in ds when them are NULL.

*Constraints*
- If *ds* is a scalar value, *ds* and *rep_value* must be equal in type between themselves.
- If *ds* is a Dataset, all its Measure Components must be of the same type and *rep_value* must be of the same type of the *ds* Measure Components.

*Returns*
If *ds* is a scalar value, it returns *ds* if it is not NULL, rep_value otherwise.
If *ds* is a Dataset, it returns a new Dataset having all the Identifier, Measure and Attribute Components of the input one, where the NULL values of the input Dataset Measure Components are replaced with the specified *rep_value*.

*Examples*
On scalar
1) If C is NULL:
A := nvl(C, 5)    A = 5
2) If COMPX is not NULL and equal to 10:
A := nvl(C, 5)    A = 10

On Dataset
3) ds_1 := **nvl(**population,0**)**

| population | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 11094850 |
| 2012 | Greece | Total | Total | 11123034 |
| 2012 | Spain | Total | Total | NULL |
| 2012 | Malta | Total | Total | 417546 |
| 2012 | Finland | Total | Total | 5401267 |
| 2012 | NULL | Total | Total | NULL |

| ds_1 | | | | |
|---|---|---|---|---|
| **TIME** | **GEO** | **AGE** | **SEX** | **POPULATION** |
| 2012 | Belgium | Total | Total | 11094850 |
| 2012 | Greece | Total | Total | 11123034 |
| 2012 | Spain | Total | Total | 0 |
| 2012 | Malta | Total | Total | 417546 |
| 2012 | Finland | Total | Total | 5401267 |
| 2012 | NULL | Total | Total | 0 |

# VTL-ML - Clause operators

## rename

*Semantics*

The **rename** operator, allows to change the name and the role of Measures or Attributes component of a dataset

*Syntax*

ds_1 **[rename** k **as** compName
{**role=**[**MEASURE|IDENTIFIER|ATTRIBUTE**]}
{, k **as** compName
{**role=**[**MEASURE|IDENTIFIER|ATTRIBUTE**]}
}***]**

*Parameters*

*ds_1* : Dataset<?>
*k* : Measure or Attribute Component
*compName* : string

*ds_1*– the input Dataset
*k* – each Component to rename
*compName* – the new name for each Component
*role* – the new role for each Component

*Returns*

The Dataset with renamed Components and changed roles.

*Constraints*

- *k* is a Component expression that can have only Component literals of ds_1 (static).
- *role* can be one of : "MEASURE", "IDENTIFIER", "ATTRIBUTE" (static).

*Semantics*

It renames each Measure or Attribute in ds_1 that is mentioned in the operator with the new name given in compName and the role given in role variable. If role variable is not specified, the role is left unmodified. All the data points in ds_1 are copied into ds_2.
Returns a new Dataset ds_2 with the same Identifier of ds_1.
The Dataset ds_2 will have the same Measure and Attributes Components of ds_1 except for those components changed in the role by the rename operator.

*Examples*

ds_2 := ds_1[rename M1 as "I1" role IDENTIFIER] 3154
The expression above renames MeasureComponent M1 into I1 and alters its role to IdentifierComponent.

## filter

*Semantics*

The operator **filter** returns the input Dataset filtered by evaluating the boolean component expression specified as a parameter.

*Syntax*

*ds* **[ filter** *f* / *dpr***]**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}*
{attribute <IDENT> as scalar-type}*
*f* : {role <IDENT> as boolean}

*dpr* : a previously defined datapoint ruleset

- *ds* – is the input Dataset.
- *f* – is a Boolean expression involving Components of *ds.*

*Constraints*

*f* is a Component expression over the Components of *ds* (static).

*Returns*

A Dataset with the same Identifier, Measure and Attribute Components of the input one, containing only the Data Points of *ds* that satisfy the Boolean expression *f* or the datapoint (horizontal) ruleset

*Semantic specification*

The operator takes as input a Dataset and a Boolean expression involving the Components owned by *ds* and returns another Dataset with the same structure of the input one. For each Data Point the expression *f* is applied; only the Data Points for which the expression is evaluated true will be part of the output Dataset.

*Examples*

Dr:=population1 **[filter** SEX = “F"**]**

| population1 | | | | |
|---|---|---|---|---|
| **SEX** | **AGE** | **GEO** | **TIME** | **POPULATION** |
| M | Y_LT15 | BE | 2013 | 970428 |
| M | Y15-64 | BE | 2013 | 3678355 |
| M | Y_GE65 | BE | 2013 | 838653 |
| F | Y_LT15 | BE | 2013 | 927644 |
| F | Y15-64 | BE | 2013 | 3625561 |
| F | Y_GE65 | BE | 2013 | 1121001 |
| M | Y_LT15 | UK | 2013 | 5757444 |
| M | Y15-64 | UK | 2013 | 20748657 |
| M | Y_GE65 | UK | 2013 | 4917238 |
| F | Y_LT15 | UK | 2013 | 5488356 |
| F | Y15-64 | UK | 2013 | 20915924 |
| F | Y_GE65 | UK | 2013 | 6068452 |

| Dr | | | | |
|---|---|---|---|---|
| **SEX** | **AGE** | **GEO** | **TIME** | **POPULATION** |
| F | Y_LT15 | BE | 2013 | 927644 |
| F | Y15-64 | BE | 2013 | 3625561 |
| F | Y_GE65 | BE | 2013 | 1121001 |
| F | Y_LT15 | UK | 2013 | 5488356 |
| F | Y15-64 | UK | 2013 | 20915924 |
| F | Y_GE65 | UK | 2013 | 6068452 |

# keep

*Semantics*

The operator **keep** returns the input Dataset with only the Identifier and Measures Components specified as parameters.

*Syntax*

*ds* **[ keep** *cmp* {, *cmp*} **]**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}*
{attribute <IDENT> as scalar-type}*
*cmp* : component-ref

- *ds* – is the input Dataset.
- *cmp* – is an existing Component of *ds.*

*Constraints*

- *cmp* is a Component expression over the Components of ds containing only Component literals (i.e. names of the Components of ds) (static).
- *cmp* cannot be a reference to an Identifier Component.

*Returns*

A Dataset having all the Identifier Components of *ds* and the Measure Components and Attribute Components selected in *cmp*.

*Semantic specification*

The operator takes as input a Dataset *ds* and a subset of the Components owned by *ds,* it returns another Dataset having all the Identifier Components of the input one (Identifier Components are not affected by the **keep**) and all the Measure Components and Attribute Components selected in *cmp.*

*Examples*

ds_1 := population1**[keep** SEX, GEO, POPULATION **]**

| population1 | | | | |
|---|---|---|---|---|
| SEX | AGE | GEO | TIME | POPULATION |
| M | Y_LT15 | BE | 2013 | 970428 |
| M | Y15-64 | BE | 2013 | 3678355 |
| M | Y_GE65 | BE | 2013 | 838653 |
| F | Y_LT15 | BE | 2013 | 927644 |
| F | Y15-64 | BE | 2013 | 3625561 |
| F | Y_GE65 | BE | 2013 | 1121001 |
| M | Y_LT15 | UK | 2013 | 5757444 |
| M | Y15-64 | UK | 2013 | 20748657 |
| M | Y_GE65 | UK | 2013 | 4917238 |
| F | Y_LT15 | UK | 2013 | 5488356 |
| F | Y15-64 | UK | 2013 | 20915924 |
| F | Y_GE65 | UK | 2013 | 6068452 |

| population1 | | |
|---|---|---|
| SEX | GEO | POPULATION |
| M | BE | 970428 |

| | | |
|---|---|---|
| M | BE | 3678355 |
| M | BE | 838653 |
| F | BE | 927644 |
| F | BE | 3625561 |
| F | BE | 1121001 |
| M | UK | 5757444 |
| M | UK | 20748657 |
| M | UK | 4917238 |
| F | UK | 5488356 |
| F | UK | 20915924 |
| F | UK | 6068452 |

# calc

*Semantic*

The operator **calc** returns the input Dataset with new components calculated based on the expressions specified as parameters.

*Syntax*

*ds* **[calc** *k* **as** *compName* {**role** [**Measure | Identifier | Attribute**]} {**viral** }
{, *k* **as** *compName* {**role [Measure | Identifier | Attribute**]} {**viral**} }*
**]**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}*
{attribute <IDENT> as scalar-type}*
*k* : expr
*compName* : ident
*role* : constant

- *ds* –is the input Dataset.
- *k* – is an expression involving ds Components.
- *role* – is the role of the calculated Component.
- *compName* – is the name of the new Component.

*Constraints*

- *role* can be one of: “Measure”, “Identifier”, “Attribute”.
- *k* is an expression on ds Components.

*Return*

A Dataset having the same Identifier and Measure Components of the input one, enriched by others Components calculated using the defined *k* expressions.

*Semantic specification*

The operator takes in as input a Dataset *ds* and a series of expressions to calculate new Components, and returns a new Dataset with the same Identifier and Measure Components.
It adds to the output Datset a Component for each Component expression *k* specified in the clause, calculating it row-wise according to the Component expression.
The added Component is named *compName* and is given a role (Identifier, Measure or Attribute Component) according to *role* Constant. If the role is omitted, “MEASURE” is implied.
If any *k* coincides with the name of an existing Component in *ds* (even with different type), the calculated one replaces the former, in name, value and type.
Special care must be paid to the handling of Attribute Components. If a Component expression has the same name as an existing Attribute Component, the previous one is overridden, independently of its virality. In this

sense, **calc** clause overrides virality. On the other hand, if no Attribute Component expressions override an existing Component, it will be kept in the result, only if viral, with unaltered virality. In general, when an Attribute Component is calculated, its virality can be set by the use of keyword viral. If it is omitted, the Attribute Component is non **viral** by default. As a special case of this, a **calc** can be also used simply to alter the virality of an Attribute Component.

*Examples*

1)

ds_2 := ds_1**[calc** M1*M2/3 **as** "M4" **role MEASURE]**

The expression above calculates a Measure Componet by combining the ones of the involved Datasets.

2)

ds_2 := ds_1**[calc** M1-1 **as** "M1**" role MEASURE,** M1+M2 **as** M2, **if** M2>3 **then** M2 **else** M3 **as** M3**]**

Like the preceding example, but with a conditional logic.

3)

ds_2 := ds_1**[calc** A1 **+** A2 **as** "A3" **role ATTRIBUTE viral]**

The expression above calculates Attribute Component A3 as a combination of A1 and A2.

# attrcalc

*Semantics*

The operator **attrcalc** returns the input Dataset with new Attribute components calculated based on the expressions specified as parameters.

*Syntax*

*ds* **[attrcalc** *k* **as** *compName* {**viral** } {, *k* **as** *compName* {**viral**} }***]**

*Parameters*

*ds* : dataset {identifier <IDENT> as scalar-type}+ {measure <IDENT> as scalar-type}*
{attribute <IDENT> as scalar-type}+

*k* : expr

*compName* : ident

- *ds* –is the input Dataset, containing Attribute Components.
- *k* – is an expression involving ds Components.
- *role* – is the role of the calculated Component.
- *compName* – is the name of the new Attribute Component.

*Constraints*

*k* is an expression on *ds* Components or on Components used to calculate *ds* properly qualified (static).

*Returns*

A Dataset with all the Identifier and Measure Components of the input one, and an Attribute Component, for each expression *k* specified, named *compName.*

*Semantic specification*

The operator takes as input a Dataset *ds* (which, in general, can be a complex expression of type Dataset) and returns a new Dataset *ds* with the same Identifier Components and Measure Components.

The output Dataset has an Attribute Component named *compName* for each Attribute Component expression *k* specified in the clause. *k* is a component expression, evaluated row-wise.

Special care must be paid to the handling of Attribute Components. If a Component expression has the same name as an existing Attribute Component, the previous one is overridden, independently of its virality. In this sense, **attrcalc** clause overrides virality. On the other hand, if no Attribute Component expressions override an existing Component, it will be kept in the result, only if viral, with unaltered virality. In general, when an Attribute Component is calculated, its virality can be set by the use of keyword **viral.** If it is omitted, the Attribute Component is non viral by default. As a special case of this, an **attrcalc** can be also used simply to alter the virality of an Attribute Component.

*Examples*

1) ds_2 := ds_1**[attrcalc** QUALITY+1 **as** QUALITY**]**

The expression calculates ds_2, keeping the QUALITY Attribute Component in ds_1, but adding 1 to its value.

| ds_1 | | | |
|---|---|---|---|
| **K1** | **K2** | **M1** | **QUALITY** |
| 1 | A | 1 | 1 |
| 2 | B | 3 | 2 |
| 3 | C | 5 | 3 |

| ds_2 | | | |
|---|---|---|---|
| **K1** | **K2** | **M1** | **QUALITY** |
| 1 | A | 1 | 2 |
| 2 | B | 3 | 3 |
| 3 | C | 5 | 4 |

2) ds_r := (ds_1 + ds_2**)[attrcalc** ds_1.QUALITY **+** ds_2.QUALITY **as** QUALITY**]**

The expression calculates ds_r as the sum of Datasets *ds_1* and *ds_2*. Besides, it calculates the QUALITY attribute as the sum of the two.

| ds_1 | | | |
|---|---|---|---|
| **K1** | **K2** | **M1** | **QUALITY** |
| 1 | A | 1 | 1 |
| 2 | B | 3 | 2 |
| 3 | C | 5 | 3 |

| ds_2 | | | |
|---|---|---|---|
| **K1** | **K2** | **M1** | **QUALITY** |
| 1 | A | 6 | 2 |
| 2 | B | 7 | 3 |
| 3 | C | 8 | 4 |

| ds_r | | | |
|---|---|---|---|
| **K1** | **K2** | **M1** | **QUALITY** |
| 1 | A | 7 | 3 |
| 2 | B | 10 | 5 |
| 3 | C | 13 | 7 |

3)

```
ds_r := (ds_1 + ds_2)
[
 attrcalc if ds_1.QUALITY="A" and ds.2QUALITY="B" then "C"
      elseif ds_1.QUALITY="K" and ds.2QUALITY="K" then "M"
       else "Z" AS AGGREGATED_QUALITY
]
```

The expression calculates the AGGREGATED_QUALITY attribute as a combination of QUALITY Attribute Components of the operands according to a decision rule.
In particular, if *ds_1* quality is "A" and *ds_2* quality is "B", then the AGGREGATED_QUALITY will be "C". Else, if *ds_1* quality is "K" and *ds_2* quality is "K", then "M" is returned. Otherwise "Z" is the AGGREGATED_QUALITY value.

| ds_1 | | | |
|---|---|---|---|
| **K1** | **K2** | **M1** | **QUALITY** |
| 1 | A | 1 | A |
| 2 | B | 3 | B |
| 3 | C | 5 | K |

| ds_2 | | | |
|---|---|---|---|
| **K1** | **K2** | **M1** | **QUALITY** |
| 1 | A | 6 | B |
| 2 | B | 7 | A |
| 3 | C | 8 | K |

| ds_r | | | |
|---|---|---|---|
| **K1** | **K2** | **M1** | **QUALITY** |
| 1 | A | 7 | C |
| 2 | B | 10 | Z |
| 3 | C | 13 | M |

4)
ds_r := (ds_1 + ds_2)**[attrcalc** ds_1.QUALITY_1 **+** ds_2.QUALITY_1 **as** QUALITY_1, ds_2.QUALITY_2 **as** QUALITY_2**]**
The expression sums two multi-measure Datasets and calculates two Attribute Components with different formulas: QUALITY_1 is the sum of *ds_1*.QUALITY_1 and *ds_2*.QUALITY_1, while QUALITY_2 is simply copied from *ds_2.*

| ds_1 | | | | | |
|---|---|---|---|---|---|
| **K1** | **K2** | **M1** | **M2** | **QUALITY_1** | **QUALITY_2** |
| 1 | A | 1 | 5 | 1 | 2 |
| 2 | B | 3 | 3 | 2 | 7 |
| 3 | C | 5 | 1 | 3 | 4 |

| ds_2 | | | | | |
|---|---|---|---|---|---|
| **K1** | **K2** | **M1** | **M2** | **QUALITY_1** | **QUALITY_2** |
| 1 | A | 6 | 1 | 2 | 1 |
| 2 | B | 7 | 1 | 3 | 3 |
| 3 | C | 8 | 1 | 4 | 1 |

| ds_r | | | | | |
|---|---|---|---|---|---|
| **K1** | **K2** | **M1** | **M2** | **QUALITY_1** | **QUALITY_2** |
| 1 | A | 7 | 6 | 3 | 1 |
| 2 | B | 10 | 4 | 5 | 3 |
| 3 | C | 13 | 2 | 7 | 1 |