SDMX Technical Working Group

VTL Task Force

VTL – version 2.0
(Validation & Transformation Language)

Part 2 – Reference Manual

April 2018

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 2.0.

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 2.0 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, described as either dimensional tables or 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.), and would provide at the same time, a "neutral" business-level expression 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 kind of 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. In this way, existing standards (GSIM, SDMX, DDI, others) would be allowed to 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 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 with the work for designing this VTL 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 draft version 1.1, which contained many new features, was completed in the second half of 2016 and provided for public consultation until the beginning of 2017.

The high number and wide impact of comments and suggestions induced a high workload on the VTL TF, which agreed to proceed in two steps for the publication of the final documentation, taking also into consideration that some first VTL implementation initiatives had already been launched. The first step, the current one, is dedicated to fixing some high-priority features and making them as much stable as possible. A second step, scheduled for the next period, is aimed at acknowledging and fixing other features considered of minor impact and priority, which will be added hopefully without affecting neither the features already published in this documentation, nor the possible relevant implementations. Moreover, taking into account the number of very important new features that have been introduced in this version in respect to the VTL 1.0, it was agreed that the current VTL version should be considered as a major one and thus named VTL 2.0.

The VTL 2.0 package contains the general VTL specifications, independently of the possible implementations of other standards; in its final release, it will include:

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 is the part 2.

The latest version of VTL is freely available online at 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, Stefano Pambianco, 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).

Table of contents

Introduction

This document is the Reference Manual of the Validation and Transformation Language (also known as 'VTL') version 2.0.

The VTL 2.0 library of the Operators is described hereinafter.

VTL 2.0 consists of two parts: the VTL Definition Language (VTL-DL) and the VTL Manipulation Language (VTL-ML).

This manual describes the operators of VTL 2.0 in detail (both VTL-DL and VTL-ML) and is organized as follows.

First, in the following Chapter "Overview of the language and conventions", the general principles of VTL are summarized, the main conventions used in this manual are presented and the operators of the VTL-DL and VTL-ML are listed. For the operators of the VTL-ML, a table that summarizes the "Evaluation Order" (i.e., the precedence rules for the evaluation of the VTL-ML operators) is also given.

The following two Chapters illustrate the operators of VTL-DL, specifically for:

- the definition of rulesets and their rules, which can be invoked with appropriate VTL-ML operators (e.g. to check the compatibility of Data Point values …);
- the definition of custom operators/functions of the VTL-ML, meant to enrich the capabilities of the VTL-ML standard library of operators.

The illustration of VTL-ML begins with the explanation of the common behaviour of some classes of relevant VTL-ML operators, towards a good understanding of general language characteristics, which we factor out and do not repeat for each operator, for the sake of compactness.

The remainder of the document illustrates each single operator of the VTL-ML and is structured in chapters, one for each category of Operators (e.g., general purpose, string, numeric …). For each Operator, there is a specific section illustrating the syntax, the semantics and giving some examples.

Overwiew of the language and conventions

Introduction

The Validation and Transformation Language is aimed at defining Transformations of the artefacts of the VTL Information Model, as more extensively explained in the User Manual.

A Transformation consists of a statement which assigns the outcome of the evaluation of an expression to an Artefact of the IM. The operands of the expression are IM Artefacts as well. A Transformation is made of the following components:

- A left-hand side, which specifies the Artefact which the outcome of the expression is assigned to (this is the result of the Transformation);
- An assignment operator, which specifies also the persistency of the left hand side. The assignment operators are two, the first one for the persistent assignment (**<-**) and the other one for the non-persistent assignment (**:=**).
- A right-hand side, which is the expression to be evaluated, whose inputs are the operands of the Transformation. An expression consists in the invocation of VTL Operators in a certain order. When an Operator is invoked, for each input Parameter, an actual argument (operand) is passed to the Operator, which returns an actual argument for the output Parameter. In the right hand side (the expression), the Operators can be nested (the output of an Operator invocation can be input of the invocation of another Operator). All the intermediate results in an expression are non-persistent.

Examples of Transformations are:

```
DS_np := ( DS_1 - DS_2 ) * 2 ;
DS_p  <- if DS_np >= 0 then DS_np else DS_1 ;
```

(DS_1 and DS_2 are input Data Sets, DS_np is a non persistent result, DS_p is a persistent result, the invoked operators (apart the mentioned assignments) are the subtraction (**-**) the multiplication (*****) the choice (**if**…**then**…**else**), the greater or equal comparison (**>=**) and the parentheses that control the order of the operators' invocations.

Like in the example above, Transformations can interact one another through their operands and results; in fact the result of a Transformation can be operand of one or more other Transformations. The interacting Transformations form a graph that is oriented and must be acyclic to ensure the overall consistency, moreover a given Artefact cannot be result of more than one Transformation (the consistency rules are better explained in the User Manual, see VTL Information Model / Generic Model for Transformations / Transformations consistency). In this regard, VTL Transformations have a strict analogy with the formulas defined in the cells of the spreadsheets.

A set of more interacting Transformations is usually needed to perform a meaningful and self-consistent task like for example the validation of one or more Data Sets. The smaller set of Transformations to be executed in the same run is called Transformation Scheme and can be considered as a VTL program.

Not necessarily Transformations need to be written in sequence like a classical software program, in fact they are associated to the Artefacts they calculate, like it happens in the spreadsheets (each spreadsheet's formula is associated to the cell it calculates).

Nothing prevents, however, from writing the Transformations in sequence, taking into account that not necessarily the Transformations are performed in the same order as they are written, because the order of execution depends on their input-output relationships (a Transformation which calculates a result that is operand of other Transformations must be executed first). For example, if the two Transformations of the example above were written in the reverse order:

```
(i)     DS_p  <- if DS_np >= 0 then DS_np else DS_1 ;
(ii)    DS_np := ( DS_1 - DS_2 ) * 2 ;
```

All the same the Transformation (ii) would be executed first, because it calculates the Data Set DS_np which is an operand of the Transformation (i).

When Transformations are written in sequence, a semicolon (;) is used to denote the end of a Transformation and the beginning of the following one.

Conventions for writing VTL Transformations

When more Transformations are written in a text, the following conventions apply.

Transformations:

- A Transformation can be written in one or more lines, therefore the end of a line does not denote the end of a Transformation.
- The end of a Tranformation is denoted by a semicolon (**;**).

Comments:

Comments can be inserted within VTL Transformations using the following syntaxes.

- A multi-line comment is embedded between **/*** and ***/** and, obviously, can span over several lines:

```
/*  multi-line
    comment text  */
```

- A single-line comment follows the symbol **//** up to the next end of line:

```
// text of a comment on a single line
```

- A sequence of spaces, tabs, end-of-line characters or comments is considered as a single space.
- The characters **/*** , ***/** , **//** and the whitespaces can be part of a string literal (within double quotes) but in such a case they are part of the string characters and do not have any special meaning.

Examples of valid comments:

Example 1:

```
/* this is a multi-line
          Comment */
```

Example 2:

```
// this is single-line comment
```

Example 3:

```
DS_r  <-   /* A is a dataset */  A +  /* B is a dataset */  B ;
```

(for the VTL this statement is the Transformation DS_r <- A + B ;)

Example 4:

```
DS_r  :=    DS_1                      // my comment
          *  DS_2 ;
```

*(for the VTL this statement is the Transformation DS_r := DS_1 * DS_2 ;)*

Typographical conventions

The Reference Manual (this manual) uses the normal font Cambria for the text and the other following typographical conventions:

Convention	Description
Italics Cambria	*Basic scalar data types (in the text)* e.g. "…must have one Identifier of type *time_period*. If the Data Set…."
Bold Arial	*Keywords (in the description of the syntax and in the text)* e.g. Rule ::={ ruleName : } { **when** antecedentCondition **then** } consequentCondition { **errorcode** errorCode } { **errorlevel** errorLevel } e.g. "…..The **rename** operator allows to rename one or more Components…"
Italics Arial	*Optional Parameter (in the description of the syntax)* e.g. **substr (** op**,** *start***,** *length* **)**
Underlined Arial	*Sub-expressions*
Normal font Arial	• *The operator's syntax (excluded the keywords, the optional Parameters and the sub-expressions)* e.g. **length (** "Hello, World!" **)** • *The examples of invocation of the operators* e.g. length ("Hello, World!") • *Optional and Mandatory Parameters (in the text)* e.g. "……If comp is a Measure in op, then in the result ….."

Abbreviations for the names of the artefacts

The names of the artefacts operated by the VTL-ML come from the VTL IM. In their turn, the names of the VTL IM artefacts are derived as much as possible from the names of the GSIM IM artefacts, as explained in the User Manual.

If the complete names are long, the VTL IM suggests also a compact name, which can be used in place of the complete name in case there is no ambiguity (for example, “Set” instead than “Value Domain Subset”, “Component” instead than “Data Set Component” and so on); moreover, to make the descriptions more compact, a number of abbreviations, usually composed of the initials (in capital case) or the first letters of the words of artefact names, are adopted in this manual:

Complete name	***Compact name***	***Abbreviation***
Data Set	*Data Set*	*DS*
Data Point	*Data Point*	*DP*
Identifier Component	*Identifier*	*Id*
Measure Component	*Measure*	*Me*
Attribute Component	*Attribute*	*At*
Data Set Component	*Component*	*Comp*
Value Domain Subset	*Subset or Set*	*Set*
Value Domain	*Domain*	*VD*

A positive integer suffix (with or without an underscore) can be added in the end to distinguish more than one instance of the same artefact (e.g., DS_1, DS_2, …, DS_N, Me1, Me2, …MeN). The suffix “r” stands for the result of a Transformation (e.g., DS_r).

Conventions for describing the operators’ syntax

Each VTL operator has an explanatory name, which recalls the operator function (e.g., “Greater than”) and a syntactical symbol, which is used to invoke the operator (e.g., “**>**”). The operator symbol may also be alphabetic, always lowercase (e.g., **round**).

In the VTL-DL, the operator symbol is the keyword **define** followed by the name of the object to be defined. The complete operator symbol is therefore a compound lowercase sentence (e.g. **define operator**).

In the VTL-ML, the operator symbol does not contain spaces and may be either a sequence of special characters (like **+**, **-**, **>=**, **<=** and so on) or a text keyword (e.g., **and**, **or**, **not**). The keyword may be compound with underscores as separators (e.g., **exists_in**).

Each operator has a syntax, which is a set of formal rules to invoke the operator correctly. In this document, the syntax of the operators is formally described by means of a meta-syntax which is not part of the VTL language, but has only presentation purposes.

The meta-syntax describes the syntax of the operators by means of *meta-expressions*, which define how the invocations of the operators must be written. The meta-expressions contain the symbol of the operator (e.g., “**join**”), the possible other keywords to denote special parameters (e.g., **using**), other symbols to be used (e.g., parentheses, commas), the named formal parameters (e.g., multiplicand and multiplier for the multiplication).

As for the typographic stile, in order to distinguish between the syntax symbols (which are used in the operator invocations) and meta-syntax symbols (used just for explanatory purposes, and not actually used in invocations), the syntax symbols are in **boldface** (i.e., the operator symbol, the special keywords, the possible parenthesis, commas an so on). The names of the generic operands (e.g., multiplicand, multiplier) are in Roman type, even if they are part of the syntax.

The meta-expression can be very simple, for example the meta-expression for the addition is:

op1 **+** op2

This means that the addition has two operands (op1, op2) and is invoked by specifying the name of the first addendum (op1), then the addition symbol (**+**) followed by the name of the second addendum (op2).

In this example, all the three parts of the meta-expression are fixed. In other cases, the meta-expression can be more complex and made of optional, alternative or repeated parts.

In the simple cases, the optional parts are denoted by using the *italic* face, for example:

substr (op**,** *start***,** *length* **)**

The expression above implies that in the **substr** operator the start and length operands are optional. In the invocation, a non-specified optional operand is substituted by an underscore or, if it is in the end of the invocation, can be omitted. Hence the following syntaxes are all formally correct:

substr (op**,** start**,** length **)**

substr (op**,** start **)**

substr (op**, _ ,** length **)**

substr (op **)**

In more complex cases, a **regular expression style** is used to denote the parts (sub-expressions) of the meta-expression that are optional, alternative or repeated. In particular, braces denote a sub-expression; a vertical bar (or sometimes named "pipe") within braces denotes possible alternatives; an optional trailing number, following the braces, specifies the number of possible repetitions.

- non-optional *: non-optional sub-expression (text without braces)*
- {optional} *: optional sub-expression (zero or 1 occurrence)*
- {non-optional}^{1} *: non-optional sub-expression (just 1 occurrence)*
- {one-or-more}+ *: sub-expression repeatable from 1 to many occurrences*
- {zero-or-more}* *: sub-expression repeatable from 0 to many occurrences*
- { part1 | part2 | part3 } *: optional alternative sub-expressions (zero or 1 occurrence)*
- { part1 | part2 | part3 }^{1} *: alternative sub-expressions (just 1 occurrence)*
- { part1 | part2 | part3 }+ *: alternative sub-expressions, from 1 to many occurrences*
- { part1 | part2 | part3 }* *: alternative sub-expressions, from 0 to many occurrences*

Moreover, to improve the readability, some sub-expressions (the underlined ones) can be referenced by their names and separately defined, for example a meta-expression can take the following form:

sub-expr$_1$-text sub-expr$_2$-name … sub-expr$_{N-1}$-name sub-expr$_N$-text

sub-expr$_2$-name ::= sub-expr$_2$-text

... possible others ...

sub-expr$_{N-1}$-name ::= sub-expr$_{N-1}$-text

In this representation of a meta-expression:

- *The first line is the text of the meta-expression*
- sub-expr$_1$-text, sub-expr$_N$-text *are sub-expressions directly written in the meta-expression*
- sub-expr$_2$-name, … sub-expr$_{N-1}$-name *are identifiers of sub-expressions.*
- sub-expr$_2$-text, … sub-expr$_{N-1}$-text *are subexpression written separately from the meta-expression.*
- *The symbol* **::=** *means "is defined as" and denotes the assignment of a sub-expression-text to a sub-expression-name.*

The following example shows the definition of the syntax of the operators for removing the leading and/or the trailing whitespaces from a string:

Meta-expression ::= { **trim | ltrim | rtrim** }^{1} **(** op **)**

The meta-expression above synthesizes that:

- **trim**, **ltrim**, **rtrim** are the operators' symbols (reserved keywords);
- **(,)** are symbols of the operators syntax (reserved keywords);
- op is the only operand of the three operators;
- "{ }^{1}" and "|" are symbols of the meta-syntax; in particular "|" indicates that the three operators are alternative (a single invocation can contain only one of them) and "{ }^{1}" indicates that a single invocation contains just one of the shown alternatives;

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

ltrim (DS_2)

rtrim (DS_3)

In these invocations, **ltrim** and **rtrim** are the symbols of the invoked operator and DS_2 and DS_3 are the names of the specific Data Sets which are operands respectively of the former and the latter invocation.

Description of the data types of operands and result

This section cointains a brief legenda of the meaning of the symbols used for describing the possible types of operands and results of the VTL operators. For a complete description of the VTL data types, see the chapter “VLT Data Types” in the User Manual.

Symbol	Meaning	Example	Example meaning
parameter **::** type2	parameter is of the *type2*	param1 :: string	param1 is of type *string*
type1 **|** type2	alternative *types*	dataset | component | scalar	either *datset* or *component* or *scalar*
type1**<**type2**>**	scalar *type2* restricts *type1*	measure<string>	Measure of *string* type
type1 _ (underscore)	*type1* can appear just once	measure<string> _	just one string Measure
type1 elementName	predetermined element of *type1*	measure<string> my_text	just one string Measure named “my_text”
type1 _ **+**	*type1* can appear one or more times	measure<string>_+	one or more string Measures
type1 _ *****	*type1* can appear zero, one or more times	measure<string>_*	zero, one or more string Measures
dataset **{** type_constraint **}**	*Type_constraint* restricts the *dataset* type	dataset { measure < string > _+ }	Dataset having one or more string Measures
$t_1 * t_2 * \ldots * t_n$	Product of the types $t_1 , t_2 , \ldots , t_n$	string * integer * boolean	triple of scalar values made of a string, an integer and a boolean value
t_1 **->** t_2	Operator from t_1 to t_2	string -> number	Operator having input string and output number
ruleset **{** type_constraint **}**	*Type_constraint* restricts the *ruleset* type	hierarchical { geo_area }	hierarchical ruleset defined on geo_area
set **<** t **>**	Set of elements of type “t”	set < dataset >	set of datasets

Moreover, the word “name” in the data type description denotes the fact that the argument of the invocation can contain only the name of an artefact of such a type but not a sub-expression. For example:

comp :: name < component < string > >

Means that the argument passed for the input parameter comp can be only the name of a Component of the basic scalar type *string*. The argument passed for comp cannot be a component expression.

The word “name” added as a suffix to the parameter name means the same (for example if the parameter above is called comp_name).

VTL-ML Operators

Name	Symbol	Syntax	Description	Notation	Input parameters type	Result type	Behaviour
Parentheses	**( )**	**(** op **)**	Override the default evaluation order of the operators	Func.	op :: dataset | component | scalar	dataset |component | scalar	Specific
Persistent assignment	**<-**	re **<-** op	Assigns an Expression to a persistent model artefact	Infix	op :: dataset	dataset	Specific
Non persistent assignment	**:=**	re **:=** op	Assigns an Expression to a non persistent model artefact	Infix	op :: dataset | scalar	dataset	Specific
Membership	**#**	ds#comp	Identifies a Component within a Data Set	Infix	ds :: dataset comp :: name<component>	dataset	Specific
User defined operator call		operator_name **(** { argument { , argument }* } **)**	Invokes a user defined operator passing the arguments	Func.	operatorName :: name argument :: user-defined operator parameters data type	user-defined result data type	Specific
Evaluation of an external routine	**eval**	**eval** **(** externalRoutineName **(** {argument} {, argument }* **)** , language, **returns** outputType **)**	Evaluates an external routine	Func.	externalRoutineName :: string argument :: any dataType language :: string outputType :: string	dataset	Specific

<table>
<tr><td>Type conversion</td><td>cast</td><td>cast (op ,scalarType { , mask })</td><td>converts to the specified data type</td><td>Func.</td><td>op :: dataset{ measure<scalar> _ }
| component<scalar>
| scalar

scalarType :: scalar type

mask :: string</td><td>dataset{ measure<scalar> _ }
| component<scalar>
| scalar</td><td>Changing data type</td></tr>
<tr><td>Join</td><td>inner_join,
left_join,
full_join,
cross_join,</td><td>joinOperator (ds1 { as alias1 }, … ,dsN { as aliasN }
{ using usingComp }
{ filter filterCondition }
{ apply applyExpr
| calc calcClause
| aggr aggrClause { groupingClause }
}
{ keep comp {, comp }*
| drop comp {, comp }* }
{ rename compFrom to compTo
{ , compFrom to compTo }* }
)
joinOperator::= { inner_join | left_join| full_join | cross_join }[1]
calcClause ::= { calcRole } calcComp := calcExpr
{ , { calcRole } calcComp := calcExpr }*
calcRole :: { identifier | measure | attribute | viral attribute} [1]
aggrClause ::= { aggrRole } aggrComp := aggrExpr
{ , { aggrRole } aggrComp := aggrExpr }*
aggrRole ::= { measure | attribute | viral attribute }[1]
groupingClause ::= { group by idList
| group except idList
| group all conversionExpr }[1]
{ having havingCondition }</td><td>Inner join,
left outer join,
full outer join,
cross join,</td><td>Func.</td><td>ds1, …, dsN :: dataset

alias1, …, aliasN :: name

usingId :: name < component >

filterCondition :: component<boolean>

applyExpr :: dataset

calcComp:: name<component>

calcExpr :: component<scalar>

aggrComp :: name<component >

aggrExpr :: component<scalar>

groupingId :: name < identifier >

conversionExpr :: component<scalar>

havingCondition :: component<boolean>

comp :: name < component >

compFrom :: component<scalar>

compTo :: component<scalar></td><td>dataset</td><td>Specific</td></tr>
<tr><td>String concatenation</td><td>||</td><td>op1 || op2</td><td>Concatenates two strings</td><td>Infix</td><td>op1, op2 ::
dataset { measure<string> _+}
| component<string>
| string</td><td>dataset { measure<string> _+ }
| component<string>
| string</td><td>On two scalars, DSs or DSCs</td></tr>
</table>

<table>
<tr><td>Whitespace removal</td><td>trim
rtrim
ltrim</td><td>{trim|ltrim|rtrim}[1] (op)</td><td>Removes trailing or/and leading whitespace from a string</td><td>Func.</td><td>op ::
dataset { measure<string> _+ }
| component<string>
| string</td><td>dataset { measure<string> _+ }
| component<string>
| string</td><td>On one scalar, DS or DSC</td></tr>
<tr><td>Character case conversion</td><td>upper
lower</td><td>{upper | lower}[1] (op)</td><td>Converts the character case of a string in upper or lower case</td><td>Func.</td><td>op ::
dataset { measure<string> _+ }
| component<string>
| string</td><td>dataset { measure<string> _+ }
| component<string>
| string</td><td>On one scalar, DS or DSC</td></tr>
<tr><td>Sub-string extraction</td><td>substr</td><td>substr (op, start, length)</td><td>Extracts the substring that starts in a specified position and has a specified lengtt</td><td>Func.</td><td>op ::
dataset { measure<string> _+ }
| component<string>
| string

start ::
component < integer[>=1]>
| integer[>= 1]

length ::
component < integer[>= 0] >
| integer[>=0]</td><td>dataset { measure<string> _+ }
| component<string>
| string</td><td>On one DS

or

on more than two scalars or DSC</td></tr>
<tr><td>String pattern replacement</td><td>replace</td><td>replace (op ,pattern1, pattern2)</td><td>Replaces a specified string-pattern with another one</td><td>Func.</td><td>op ::
dataset { measure<string> _+ }
| component<string>
| string

pattern1, pattern2 ::
component<string>
| string</td><td>dataset { measure<string> _+ }
| component<string>
| string</td><td>On one DS

or

on more than two scalars or DSC</td></tr>
</table>

String pattern location	**instr**	**instr(** op, pattern, *start*, *occurrence* **)**	Returns the location of a specified string-pattern	Func.	op :: dataset { measure<string> _+ } | component<string> | string pattern :: component<string> | string start:: component< integer[>= 1]> | integer[>= 1] occurrence :: component < integer[>= 1] > | integer[>= 1]	dataset {measure<integer[>=0]> int_var } | component <integer[>= 0]> | integer[>= 0]	Changing data type
String length	**length**	**length (** op **)**	Returns the length of a string	Func.	op :: dataset { measure<string> _ } | component<string> | string	dataset {measure<integer[>=0]> int_var } | component <integer[>= 0]> | integer[>= 0]	Changing data type
Unary plus	**+**	**+** op	Replicates the operand with the sign unaltered	Infix	op :: dataset { measure<number> _+ } | component<number> | number	dataset { measure<number> _+ } | component<number> | number	On one scalar, DS or DSC
Unary minus	**-**	**-** op	Replicates the operand with the sign changed	Infix	op :: dataset { measure<number> _+ } | component<number> | number	dataset { measure<number> _+ } | component<number> | number	On one scalar, DS or DSC
Addition	**+**	op1 **+** op2	Sums two numbers	Infix	op1, op2:: dataset { measure<number> _+ } | component<number> | number	dataset { measure<number> _+ } | component<number> | number	On two scalars, DSs or DSCs
Subtraction	**-**	op1 **-** op2	Subtracts two numbers	Infix	op1, op2:: dataset { measure<number> _+ } | component<number> | number	dataset { measure<number> _+ } | component<number> | number	On two scalars, DSs or DSCs
Multiplication	*	op1 * op2	Multiplies two numbers	Infix	op1, op2:: dataset { measure<number> _+ } | component<number> | number	dataset { measure<number> _+ } | component<number> | number	On two scalars, DSs or DSCs
Division	**/**	op1 **/** op2	Divides two numbers	Infix	op1, op2:: dataset { measure<number> _+ } | component<number> | number	dataset { measure<number> _+ } | component<number> | number	On two scalars, DSs or DSCs

Modulo	**mod**	**mod (** op1, op2**)**	Calculates the remainder of a number divided by a certain divisor	Func.	op1, op2:: dataset { measure<number> _+ } | component<number> | number	dataset { measure<number> _+ } | component<number> | number	On two scalar, DS or DSC
Rounding	**round**	**round (** op, *numDigit* **)**	Rounds a number to a certain digit	Func.	op :: dataset { measure<number> _+ } | component<number> | number numDigit:: component < integer > | integer	dataset { measure<number> _+ } | component<number> | number	On one DS or on two scalars or DSC
Truncation	**trunc**	**trunc (** op, *numDigit* **)**	Truncates a number to a certain digit	Func.	op :: dataset { measure<number> _+ } | component<number> | number numDigit :: component < integer > | integer	dataset { measure<number> _+ } | component<number> | number	On one DS or on two scalars or DSC
Ceiling	**ceil**	**ceil (** op **)**	Returns the smallest integer which is greater or equal than a number	Func.	op :: dataset { measure<number> _+ } | component<number> | number	dataset { measure<integer> _+ } | component< integer > | integer	On one scalar, DS or DSC
Floor	**floor**	**floor (** op **)**	Returns the greater integer which is smaller or equal than a number	Func.	op :: dataset { measure<number> _+ } | component<number> | number	dataset { measure<integer> _+ } | component< integer > | integer	On one scalar, DS or DSC
Absolute value	**abs**	**abs (** op **)**	Calculates the absolute value of a number	Func.	op :: dataset { measure<number> _+ } | component<number> | number	dataset { measure<number[>=0]> _+ } | component<number[>=0]> | number[>= 0]	On one scalar, DS or DSC
Exponential	**exp**	**exp (** op **)**	Raises e (base of the natural logarithm) to a number	Func.	op:: dataset { measure<number> _+ } | component<number> | number	dataset { measure<number[>0]> _+ } | component<number[>0]> | number[> 0]	On one scalar, DS or DSC

<table>
<tr><td>Natural logarithm</td><td>ln</td><td>ln (op)</td><td>Calculates the natural logarithm of a number</td><td>Func.</td><td>op ::
dataset
{measure<number[>0]> _+ }
| component<number[>0]>
| number[>0]</td><td>dataset
{ measure<number> _+ }
| component<number>
| number</td><td>On one scalar, DS or DSC</td></tr>
<tr><td>Power</td><td>power</td><td>power (base, exponent)</td><td>Raises a number to a certain exponent</td><td>Func.</td><td>base ::
dataset { measure<number> _+ }
| component<number>
| number

exponent ::
component<number> | number</td><td>dataset
{ measure<number> _+ }
| component<number>
| number</td><td>On one DS

or

on two scalars or DSC</td></tr>
<tr><td>Logarithm</td><td>log</td><td>log (op, num)</td><td>Calculates the logarithm of a number to a certain base</td><td>Func.</td><td>op :: dataset
{ measure<number[>1]> _+ }
| component<number[>1]>
| number[>1]

num:: component<integer[>0]>
| integer[>0]</td><td>dataset
{ measure<number> _+ }
| component<number>
| number</td><td>On one DS

or

on two scalars or DSC</td></tr>
<tr><td>Square root</td><td>sqrt</td><td>sqrt (op)</td><td>Calculates the square root of a number</td><td>Func.</td><td>op :: dataset
{ measure<number[>=0> _+ }
| component<number[>= 0]>
| number[>= 0]</td><td>dataset
{ measure<number[>=0]> _+ }
| component<number[>= 0]>
| number[>= 0]</td><td>On one scalar, DS or DSC</td></tr>
<tr><td>Equal to</td><td>=</td><td>left = rigth</td><td>Verifies if two values are equal</td><td>Infix</td><td>left,right ::
dataset {measure<scalar> _ }
| component<scalar>
| scalar</td><td>dataset
{measure<boolean> bool_var}
| component<boolean>
| boolean</td><td>Changing data type</td></tr>
<tr><td>Not equal to</td><td><></td><td>left <> rigth</td><td>Verifies if two values are not equal</td><td>Infix</td><td>left, right ::
dataset {measure<scalar> _ }
| component<scalar>
| scalar</td><td>dataset
{measure<boolean> bool_var}
| component<boolean>
| boolean</td><td>Changing data type</td></tr>
<tr><td rowspan="2">Greater than</td><td>></td><td rowspan="2">left { > | >= }[1] right</td><td rowspan="2">Verifies if a first value is greater (or equal) than a second value</td><td rowspan="2">Infix</td><td rowspan="2">left, right ::
dataset {measure<scalar> _ }
| component<scalar>
| scalar</td><td rowspan="2">dataset
{measure<boolean> bool_var}
| component<boolean>
| boolean</td><td rowspan="2">Changing data type</td></tr>
<tr><td>>=</td></tr>
<tr><td rowspan="2">Less than</td><td><</td><td rowspan="2">left { < | <= }[1] right</td><td rowspan="2">Verifies if a first value is less (or equal) than a second value</td><td rowspan="2">Infix</td><td rowspan="2">left, right ::
dataset {measure<scalar> _ }
| component<scalar>
| scalar</td><td rowspan="2">dataset
{measure<boolean> bool_var}
| component<boolean>
| boolean</td><td rowspan="2">Changing data type</td></tr>
<tr><td><=</td></tr>
</table>

Between	**between**	**between(** op, from, to **)**	Verify if a value belongs to a range of values	Func.	op :: dataset {measure<scalar> _} | component<scalar> | scalar from ::scalar | component<scalar> to :: scalar | component<scalar>	dataset {measure<boolean> bool_var} | component<boolean> | boolean	Changing data type
Element of	**in**	op **in** collection **collection** ::= set | valueDomainName	Verifies if a value belongs to a set of values	Infix	op :: dataset {measure<scalar> _ } | component<scalar> | scalar collection :: set<scalar> | name<value_domain>	dataset {measure<boolean> bool_var} | component<boolean> | boolean	Changing data type
	not_in	op **not_in** collection collection ::= set | valueDomainName	Verifies if a value does not belong to a set of values	Infix			
Match_characters	**match_characters**	**match_characters (**op, pattern**)**	Verifies if a value respects or not a pattern	Func.	op:: dataset {measure<string> _} | component<string> | string pattern :: string | component<string>	dataset {measure<boolean> bool_var} | component<boolean> | boolean	Changing data type
Isnull	**isnull**	**isnull (** op **)**	Verifies if a values is NULL	Func.	op :: dataset {measure<scalar> _} | component<scalar> | scalar	dataset {measure<boolean> bool_var} | component<boolean> | boolean	Changing data type
Exists in	**exists_in**	**exists_in (** op1, op2, retain**)** retain := { **true** | **false** | **all** }	As for the common identifiers of op1 and op2, verifies if the combinations of values of op1 exist in op2.	Func.	op1, op2 :: dataset	dataset {measure<boolean> bool_var}	Changing data type
Logical conjunction	**and**	op1 **and** op2	Calculates the logical AND		op1,op2 :: dataset {measure<boolean> _ } | component<boolean> | boolean	dataset { measure<boolean> _} | component<boolean> | boolean	Boolean
Logical disjunction	**or**	op1 **or** op2	Calculates the logical OR		op1,op2 :: dataset {measure<boolean> _ } | component<boolean> | boolean	dataset { measure<boolean> _} | component<boolean> | boolean	Boolean

Exclusive disjunction	**xor**	op1 **xor** op2	Calculates the logical XOR		op1,op2 :: dataset {measure<boolean> _ } | component<boolean> | boolean	dataset { measure<boolean> _} | component<boolean> | boolean	Boolean
Logical negation	**not**	**not** op	Calculates the logical NOT		op :: dataset {measure<boolean> _ } | component<boolean> | boolean	dataset { measure<boolean> _ } | component<boolean> | boolean	Boolean
Period indicator	**period_indicator**	**period_indicator (** {op} **)**	extracts the period indicator from a time_period value	Func.	op :: dataset { identifier <time_period> _ , identifier _* } | component<time_period> | time_period	dataset { measure<duration> duration_var } | component <duration> | duration	Specific
Fill time series	**fill_time_series**	**fill_time_series (** op { , <u>limitsMethod</u> } **)** <u>limitsMethod</u> ::= **single** | **all**	Replaces each missing data point in the input Data Set	Func.	op :: dataset { identifier <time> _ , identifier _* }	dataset { identifier <time> _ , identifier _* }	Specific
Flow to stock	**flow_to_stock**	**flow_to_stock (** op **)**	Transforms from a flow interpretation of a Data Set to stock	Func.	op :: dataset { identifier <time> _ , identifier _* , measure<number> _+ }	dataset { identifier < time > _ , identifier _* , measure<number> _+ }	Specific
Stock to flow	**stock_to_flow**	**stock_to_flow (** op **)**	Transforms from stock to flow interpretation of a Data Set	Func.	op :: dataset { identifier <time> _ , identifier _* , measure<number> _+ }	dataset { identifier < time > _ , identifier _* , measure<number> _+ }	Specific
Time shift	**timeshift**	**timeshift (** op , shiftNumber **)**	Shifts the time component of a specified range of time	Func.	op :: dataset { identifier <time> _ , identifier _* } shiftNumber :: integer	dataset { identifier < time > _ , identifier _* }	Specific
Time aggregation	**time_agg**	**time_agg (** periodIndTo { , periodIndFrom } { ,op }{ , **first** | **last** }**)**	converts the time values from higher to lower frequency values	Func.	op :: dataset { identifier <time> _ , identifier _* } | component<time> | time periodIndFrom :: duration periodIndTo :: duration	dataset { identifier < time > _ , identifier _* } | component<time> | time	Specific

Actual time	**current_ date**	**current_date ( )**	returns the current date	Func.		date	Specific
Union	**union**	**union (** dsList **)** dsList ::= ds { , ds }*	Computes the union of N datasets	Func.	ds :: dataset	dataset	Set
Intersection	**intersect**	**intersect (** dsList **)** dsList ::= ds { , ds }*	Computes the intersection of N datasets	Func.	ds :: dataset	dataset	Set
Set difference	**setdiff**	**setdiff (** ds1, ds2 **)**	Computes the differences of two datasets	Func.	ds1, ds2 :: dataset	dataset	Set
Simmetric difference	**symdiff**	**symdiff (** ds1, ds2 **)**	Computes the symmetric difference of two datasets	Func.	ds1, ds2 :: dataset	dataset	Set
Hierarchical roll-up	**hierarch y**	**hierarchy (** op , hr { **condition** condComp { **,** condComp }* } { **rule** ruleComp } { mode } { input } { output }) condComp ::= component { , component }* mode ::= **non_null** | **non_zero** | **partial_null** | **partial_zero** | **always_null** | **always_zero** input ::= **dataset** | **rule** | **rule_priority** output ::= **computed** | **all**	Aggregates data using a hierarchical ruleset	Func.	op ::dataset{measure<number> _ } hr ::name < hierarchical > condComp :: name < component > ruleComp :: name < identifier >	dataset{measure<number> _ }	Specific
Aggregate invocation		*in a Data Set expression:* aggregateOperator **(** firstOperand { , additionalOperand }* { groupingClause } **)** *in a Component expression within an aggr clause* aggregateOperator **(** firstOperand { , additionalOperand }* **)** { groupingClause } aggregateOperator ::= **avg** | **count** | **max** | **median** | **min** | **stddev_pop**| **stddev_samp** | **sum** | **var_pop** | **var_samp** groupingClause ::= { **group by** groupingId {, groupingId}* | **group except** groupingId {, groupingId}* | **group all** conversionExpr }1 { **having** havingCondition }	Set of statistical functions used to aggregate data	Func.	firstOperand :: dataset | component additionalOperand :: type of the (possible) additional parameter of the aggregate Operator groupingId ::name < identifier > conversionExpr :: identifier havingCondition :: component<boolean>	dataset | component	Specific

Analytic invocation		analyticOperator **(** firstOperand { , additionalOperand }* **over** (<u>analyticClause</u>) **)** analyticOperator ::= **avg | count | max | median | min | stddev_pop| stddev_samp | sum | var_pop | var_samp | first_value | lag | last_value | lead | rank | ratio_to_report** <u>analyticClause</u> ::= { <u>partitionClause</u> } { <u>orderClause</u> } { <u>windowClause</u> } <u>partitionClause</u> ::= **partition by** identifier { , identifier }* <u>orderClause</u> ::= <u>order by</u> component { **asc | desc** } { , component { **asc | desc** } }* <u>windowClause</u> ::= { **data points | range** }[1] **between** <u>limitClause</u> **and** <u>limitClause</u> <u>limitClause</u> ::= { num **preceding** | num **following | current data point** | **unbounded preceding | unbounded following** }[1]	Set of statistical functions used to aggregate data	Func.	firstOperand :: dataset | component additionalOperand :: type of the (possible) additional parameter of the invoked operator identifier :: name<identifier> component :: name<component> num :: integer	dataset | component	Specific
Check datapoint	**check_datapoint**	**check_datapoint** **(** op , dpr { **components** <u>listComp</u> } { **output** <u>output</u> }) <u>listComp</u> ::= comp { , comp }* <u>output</u> ::= **invalid | all | all_measures**	Applies one datapoint ruleset on a Data Set	Func.	op ::dataset dpr ::name < datapoint > comp :: name < component >	dataset	Specific
Check hierarchy	**check_hierarchy**	**check_hierarchy (** op , hr { **condition** condComp { **,** condComp }* } { **rule** ruleComp } { <u>mode</u> } { <u>input</u> } { <u>output</u> } **)** <u>mode</u> ::= **non_null | non_zero | partial_null | partial_zero | always_null | always_zero** <u>input</u> ::= **dataset | dataset_priority** <u>output</u> ::= **invalid | all | all_measures**	Applies a hierarchical ruleset to a Data Set	Func.	op ::dataset hr ::name < hierarchical > condComp :: name< component > ruleComp :: name< identifier >	dataset	Specific
Check	**check**	**check (** op { **errorcode** errorcode } { **errorlevel** errorlevel } { **imbalance** imbalance } { <u>output</u> } **)** <u>output</u> ::= **invalid | all**	Checks if an expression verifies a condition	Func.	op :: dataset errorcode :: errorcode_vd errorlevel :: errorlevel_vd imbalance :: number	dataset	Specific

If then else	**if ....then else....**	**if** condition **then** thenOperand **else** elseOperand	Makes alternative calculations according to a condition	Func.	condition :: dataset { measure <boolean> _ } | component<boolean> | boolean thenOperand :: dataset | component | scalar elseOperand :: dataset | component | scalar	dataset | component | scalar	Specific
Nvl	**nvl**	**nvl (** op1, op2 **)**	Replaces the null value with a value.	Func.	op1, op2:: dataset | component | scalar	dataset | component | scalar	Specific
Filtering Data Points	**filter**	op **[filter** condition **]**	Filter data using a Boolean condition	Clause	op :: dataset filterCondition :: component<boolean>	dataset	Specific
Calculation of a Component	**calc**	op **[calc** { calcRole } calcComp := calcExpr { , { calcRole } calcComp := calcExpr }* **]**	Calculates the values of a Structure Component	Clause	op :: dataset calcComp ::name < component > calcExpr :: component<scalar>	dataset	Specific
Aggregation	**aggr**	op **[aggr** aggrClause { groupingClause } **]** **aggrClause** ::= { aggrRole } aggrComp := aggrExpr { , { aggrRrole } aggrComp:= aggrExpr }* groupingClause ::= { **group by** groupingId {, gropuingId }* | **group except** groupingId {, groupingId }* | **group all** conversionExpr }[1] { **having** havingCondition } aggrRole::= **measure** | **attribute** | **viral attribute**	Aggregates using an aggregate operator	Clause	op :: dataset aggrComp :: name < component > aggrExpr :: component<scalar> groupingId ::name <identifier > conversionExpr :: identifier<scalar> havingCondition :: component<boolean>	dataset	Specific
Maintaining Components	**keep**	op **[keep** comp {, comp }* **]**	Keep list of components	Clause	op ::dataset comp :: name < component >	dataset	Specific
Removal of Components	**drop**	op **[drop** comp { , comp }* **]**	Drop list of components	Clause	op :: dataset comp :: name < component >	dataset	Specific

Change of Component name	**rename**	op **[rename** comp_from **to** comp_to { ,comp_from **to** comp_to }* **]**	Rename components	Clause	op :: dataset comp_from :: name<component> comp_to :: name<component>	dataset	Specific
Pivoting	**pivot**	op **[pivot** identifier , measure **]**	Transform identifier values to measures	Clause	op :: dataset identifier ::name <identifier> measure ::name <measure>	dataset	Specific
Unpivoting	**unpivot**	op **[unpivot** identifier , measure **]**	Transform measures to identifier values	Clause	op :: dataset identifier :: name<identifier> measure :: name<measure>	dataset	Specific
Subspace	**sub**	op **[sub** identifier = value { , identifier = value }* **]**	Remove the specified identifiers by fixing a value for them	Clause	op :: dataset identifier :: name<identifier> value :: scalar	dataset	Specific

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 the default associativity rule. Precedence and associativity orders are reported in the following table.

<table>
<tr><th>Evaluation order</th><th>Operator</th><th>Description</th><th>Default associativity rule</th></tr>
<tr><td>I</td><td>( )</td><td>Parentheses. To alter the default order.</td><td>None</td></tr>
<tr><td>II</td><td>VTL operators with functional syntax</td><td>VTL operators with functional syntax</td><td>Left-to-right</td></tr>
<tr><td>III</td><td>Clause
Membership</td><td>Clause
Membership</td><td>Left-to-right</td></tr>
<tr><td>IV</td><td>unary plus
unary minus
not</td><td>Unary minus
Unary plus
Logical negation</td><td>None</td></tr>
<tr><td>V</td><td>*
/</td><td>Multiplication
Division</td><td>Left-to-right</td></tr>
<tr><td>VI</td><td>+
-
||</td><td>Addition
Subtraction
String concatenation</td><td>Left-to-right</td></tr>
<tr><td>VII</td><td>> >=
< <=
=
<>
in
not_in</td><td>Greater than
Less than
Equal-to
Not-equal-to
In a value list
Not in a value list</td><td>Left-to-right</td></tr>
<tr><td>VIII</td><td>and</td><td>Logical AND</td><td>Left-to-right</td></tr>
<tr><td>IX</td><td>or
xor</td><td>Logical OR
Logical XOR</td><td>Left-to-right</td></tr>
<tr><td>X</td><td>if-then-else</td><td>Conditional (if-then-else)</td><td>None</td></tr>
</table>

Description of VTL Operators

The structure used for the description of the VTL-DL Operators is made of the following parts:

- **Operator name**, which is also used to invoke the operator
- **Semantics:** a brief description of the purpose of the operator
- **Syntax:** the syntax of the Operator (this part follows the conventions described in the previous section “Conventions for describing the operators’ syntax”)
- **Syntax description:** detailed explanation of the meaning of the various parts of the syntax
- **Parameters:** list of the input parameters and their types

- **Constraints**: additional constraints that are not specified with the meta-syntax and need a textual explanation
- **Semantic specifications**: detailed description of the semantics of the opoerator
- **Examples**: examples of invocation of the operator

The structure used for the description of the VTL-ML Operators is made of the following parts:

- **Operator name**, followed by the **operator symbol** (keyword) which is used to invoke the operator
- **Syntax:** the syntax of the Operator (this part follows the conventions described in the previous section "Conventions for describing the operators' syntax")
- **Input parameters:** list of all input parameters and the subexpressions with their meaning and the indication if they are mandatory or optional
- **Examples of valid syntaxes:** examples of syntactically valid invocations of the Operator
- **Semantics for scalar operations**: the behaviour of the Operator on scalar operands, which is the basic behaviour of the Operator
- **Input parameters type**: the formal description of the type of the input parameters (this part follows the conventions described in the previous section "Description of the data types of operands and results")
- **Result type:** the formal description of the type of the result (this part follows the conventions described in the previous section "Description of the data types of operands and results")
- **Additional constraints**: additional constraints that are not specified with the meta-syntax and need a textual explanation, including both possible semantic constraints under which the operation is possible or impossible, and syntactical constraint for the invocation of the Operator
- **Behaviour**: description of the behaviour of the Operator for non-scalar operations (for example operations at Data Set or at Component level). When the Operator belongs to a class of Operators having a common behaviour, the common behavior is described once for all in a section of the chapter "Typical behaviours of the ML Operators" and therefore this part describes only the specific aspect of the behaviour and contains a reference to the section where the common part of the behaviour is described.
- **Examples**: a series of examples of invocation and application of the operator in case of operations at Data Sets or at Component level.

VTL-DL - Rulesets

define datapoint ruleset

Semantics

The Data Point Ruleset contains Rules to be applied to each individual Data Point of a Data Set for validation purposes. These rulesets are also called "horizontal" 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 individual Data Points (rows), i.e., horizontally on the tabular representation.

Syntax

define datapoint ruleset rulesetName **(** dpRulesetSignature **) is**
    dpRule
    { **;** dpRule }*
end datapoint ruleset

dpRulesetSignature ::=	**valuedomain** listValueDomains | **variable** listVariables
listValueDomains ::=	valueDomain { **as** vdAlias } { **,** valueDomain { **as** vdAlias } }*
listVariables ::=	variable { **as** varAlias } { **,** variable { **as** varAlias } }*
dpRule ::=	{ ruleName **:** } { **when** antecedentCondition **then** } consequentCondition { **errorcode** errorCode } { **errorlevel** errorLevel }

Syntax description

rulesetName	the name of the Data Point Ruleset to be defined.
dpRulesetSignature	the Cartesian space of the Ruleset (signature of the Ruleset), which specifies either the Value Domains or the Represented Variables (see the information model) on which the Ruleset is defined. If **valuedomain** is specified then the Ruleset is applicable to the Data Sets having Components that take values on the specified Value Domains. If **variable** is specified then the Ruleset is applicable to Data Sets having the specified Variables as Components.
valueDomain	a Value Domain on which the Ruleset is defined.
vdAlias	an (optional) alias assigned to a Value Domain and valid only within the Ruleset, this can be used for the sake of compactness in writing the Rules. If an alias is not specified then the name of the Value Domain (parameter valueDomain) is used in the body of the rules.
variable	a Represented Variable on which the Ruleset is defined.
varAlias	an (optional) alias assigned to a Variable and valid only within the Ruleset, this can be used for the sake of compactness in writing the Rules. If an alias is not specified then the name of the Variable (parameter valueDomain) is used in the body of the Rules.
dpRule	a Data Point Rule, as defined in the following parameters.
ruleName	the name assigned to the specific Rule within the Ruleset. If the Ruleset is used for validation then the ruleName identifies the validation results of the various Rules of the Ruleset. The ruleName is optional and, if not specified, is assumed to be the progressive order number of the Rule in the Ruleset. However please note that, if ruleName is omitted, then the Rule names can change in case the Ruleset is modified, 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.
antecedentCondition	a *boolean* expression to be evaluated for each single Data Point of the input Data Set. It can contain Values of the Value Domains or Variables specified in the Ruleset signature and constants; all the VTL-ML component level operators are allowed. If omitted then antecedentCondition is assumed to be TRUE.
consequentCondition	a *boolean* expression to be evaluated for each single Data Point of the input Data Set when the antecedentCondition evaluates to TRUE (as mentioned, missing antecedent

	conditions are assumed to be TRUE). It contains Values of the Value Domains or Variables specified in the Ruleset signature and constants; all the VTL-ML component level operators are allowed. A consequent condition equal to FALSE is considered as a non-valid result.
errorCode	a literal denoting the error code associated to the rule, to be assigned to the possible non-valid results in case the Rule is used for validation. If omitted then no error code is assigned (NULL value). VTL assumes that a Value Domain errorcode_vd of error codes exists in the Information Model and contains all possible error codes: the errorCode literal must be one of the possible Values of such a Value Domain. VTL assumes also that a Variable errorcode for describing the error codes exists in the IM and is a dependent variable of the Data Sets which contain the results of the validation.
errorLevel	a literal denoting the error level (severity) associated to the rule, to be assigned to the possible non-valid results in case the Rule is used for validation. If omitted then no error level is assigned (NULL value). VTL assumes that a Value Domain errorlevel_vd of error levels exists in the Information Model and contains all possible error levels: the errorLevel literal must be one of the possible Values of such a Value Domain. VTL assumes also that a Variable errorlevel for describing the error levels exists in the IM and is a dependent variable of the Data Sets which contain the results of the validation.

Parameters

rulesetName ::	name <ruleset >
valueDomain ::	name < valuedomain >
vdAlias ::	name
variable ::	name
varAlias ::	name
ruleName ::	name
antecedentCondition ::	boolean
consequentCondition ::	boolean
errorCode ::	errorcode_vd
errorLevel ::	errorlevel_vd

Constraints

- antecedentCondition and consequentCondition can refer only to the Value Domains or Variables specified in the dpRulesetSignature.
- Either ruleName is specified for all the Rules of the Ruleset or for none.
- If specified, then ruleName must be unique within the Ruleset.

Semantic specification

This operator defines a persistent Data Point Ruleset named rulesetName that can be used for validation purposes.

A Data Point Ruleset is a persistent object that contains Rules to be applied to the Data Points of a Data Set[1]. The Data Point Rulesets can be invoked by the **check_datapoint** operator. The Rules are aimed at checking the combinations of values of the Data Set Components, assessing if these values fulfil the logical conditions expressed by the Rules themselves. The Rules are evaluated independently for each Data Point, returning a Boolean scalar value (i.e., TRUE for valid results and FALSE for non-valid results).

Each Rule contains an (optional) antecedentCondition *boolean* expression followed by a consequentCondition *boolean* expression and expresses a logical implication. Each Rule states that when the antecedentCondition evaluates to TRUE for a given Data Point, then the consequentCondition is expected to be TRUE as well. If this implication is fulfilled, the result is considered as valid (TRUE), otherwise as non-valid (FALSE). On the other side, if the antecedentCondition evaluates to FALSE, the consequentCondition does not applies and is not evaluated at all, and the result is considered as valid (TRUE). In case the antecedentCondition is absent then it is assumed to be always TRUE, therefore the consequentCondition is expected to evaluate to TRUE for all the Data Points. See an example below:

[1] In order to apply the Ruleset to more Data Sets, these Data Sets must be composed together using the appropriate VTL operators in order to obtain a single Data Set.

Rule	*Meaning*
On Value Domains: **when** flow_type **=** "CREDIT" or flow_type **=** "DEBIT" **then** numeric_value **>=** 0	When the Component of the Data Set which is defined on the Value Domain named flow_type takes the value “CREDIT” or the value “DEBIT”, then the other Component defined on the Value Domain named numeric_value is expected to have a zero or positive value.
On Variables: **when** flow **=** "CREDIT" or flow = "DEBIT" **then** obs_value **>= 0**	When the Component of the Data Set named flow has the value “CREDIT” or “DEBIT” then the Component named obs_value is expected to have a value greater than zero.

The definition of a Ruleset comprises a **signature** (dpRulesetSignature), which specifies the Value Domains or Variables on which the Ruleset is defined and a set of Rules, that are the Boolean expressions to be applied to each Data Point. The antecedentCondition and consequentCondition of the Rules can refer only to the Value Domains or Variables of the Ruleset signature.
The Value Domains or the Variables of the Ruleset signature identify the space in which the rules are defined while each Rule provides for a criterion that demarcates the Set of valid combinations of Values inside this space.
The Data Point Rulesets can be defined in terms of Value Domains in order to maximize their reusability, in fact this way a Ruleset can be applied on any Data Set which has Components which take values on the Value Domains of the Ruleset signature. The association between the Components of the Data Set and the Value Domains of the Ruleset signature is provided by the **check_datapoint** operator at the invocation of the Ruleset.
When the Ruleset is defined on Variables, their reusability is intentionally limited to the Data Sets which contains such Variables (and not to other possible Variables which take values from the same Value Domain). If at a later stage the Ruleset would need to be applied also to other Variables defined on the same Value Domain, a similar Ruleset should be defined also for the other Variable.
Rules are uniquely identified by ruleName. If omitted then ruleName is implicitly assumed to be the progressive order number of the Rule in the Ruleset. Please note however that, using this default mechanism, the Rule Name can change if the Ruleset is modified, 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. In addition, if the results of more than one Ruleset have to be combined in one Data Set, then the user should make the relevant rulesetNames different.
As said, each Rule is applied in a row-wise fashion to each individual Data Point of a Data Set. The references to the Value Domains defined in the antecedentCondition and consequentCondition are replaced with the values of the respective Components of the Data Point under evaluation.
.

Examples

```
define datapoint ruleset DPR_1 ( valuedomain flow_type  A, numeric_value B ) is
        when A = “CREDIT” or A = “DEBIT” then B >= 0 errorcode “Bad value” errorlevel 10
end datapoint ruleset

define datapoint ruleset DPR_2 ( variable flow F, obs_value O) is
        when F = “CREDIT” or F = “DEBIT” then O >= 0 errorcode “Bad value”
end datapoint ruleset
```

define hierarchical ruleset

Semantics

This operator defines a persistent Hierarchical Ruleset that contains Rules to be applied to individual Components of a given Data Set 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 main purpose of the hierarchical Rules is to express some more aggregated Code Items (e.g. the continents) in terms of less aggregated ones (e.g., their countries) by using Code Item Relationships. 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). These rules can be used also for validation, 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), provided that the input Data Set contains all of them, i.e. the more and the less aggregated Code Items.

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 more detailed one (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. These relationships can be recursive, i.e., the aggregated Code Items can be in their turn component of even more aggregated ones, without limitations about the number of recursions.

As a first simple example, the following Hierarchical Ruleset named "BeneluxCountriesHierarchy" 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 rule Geo_Area ) is
        BENELUX = BELGIUM + LUXEMBOURG + NETHERLANDS
end hierarchical ruleset
```

Syntax

define hierarchical ruleset rulesetName **(** hrRulesetSignature **) is**
hrRule
{ **;** hrRule }*
end hierarchical ruleset

hrRulesetSignature ::= vdRulesetSignature | varRulesetSignature

vdRulesetSignature ::= **valuedomain** { **condition** vdConditioningSignature } **rule** ruleValueDomain

vdConditioningSignature ::= condValueDomain { **as** vdAlias } { **,** condValueDomain { **as** vdAlias } }*

varRulesetSignature ::= **variable** { **condition** varConditioningSignature } **rule** ruleVariable

varConditioningSignature ::= condVariable { **as** vdAlias } { **,** condVariable { **as** vdAlias } }*

hrRule ::= { ruleName **:** } codeItemRelation { **errorcode** errorCode } { **errorlevel** errorLevel }

codeItemRelation ::=
{ **when** leftCondition **then** }
leftCodeItem { **=** | **>** | **<** | **>=** | **<=** }^{1}
{ **+** | **-** } rightCodeItem { **[** rightCondition **]** }
{ { **+** | **-** }^{1} rightCodeItem { **[** rightCondition **]** } }*

Syntax description

rulesetName	the name of the Hierarchical Ruleset to be defined.
hrRulesetSignature	the signature of the Ruleset. It specifies the Value Domain or Variable on which the Ruleset is defined, and the Conditioning Signature.
vdRulesetSignature	the signature of a Ruleset defined on Value Domains
varRulesetSignature	the signature of a Ruleset defined on Variables
hrRule	a single hierarchical rule, as described below.

vdConditioningSignature	specifies the Value Domains on which the conditions are defined. The Ruleset is meant to be applicable to the Data Sets having Components that take values on the Value Domain on which the ruleset is defined (i.e., ruleValueDomain) and on the conditioning Value Domains (i.e., condValueDomain).
ruleValueDomain	the Value Domain on which the Ruleset is defined
condValueDomain	a conditioning Value Domain of the Ruleset
vdAlias	an (optional) alias assigned to a Value Domain and valid only within the Ruleset, this can be used for the sake of compactness in writing leftCondition and rightCondition. If an alias is not specified then the name of the Value Domain (i.e., condValueDomain) must be used.
varConditioningSignature	the signature of the (possible) conditions of the Ruleset defined on Variables. It specifies the Represented Variables (see the information model) on which these conditions are defined. The Ruleset is meant to be applicable to any Data Set having Components which are defined by the Variable on which the Ruleset is expressed (i.e., variable) and on the Conditioning Variables.
ruleVariable	the variable on which the Ruleset is defined
condVariable	a conditioning Variable of the Ruleset
varAlias	an (optional) alias assigned to a Variable and valid only within the Ruleset, this can be used for the sake of compactness in writing leftCondition and rightCondition. If an alias is not specified then the name of the Variableomain (parameter condVariable) must be used.
ruleName	the name assigned to the specific Rule within the Ruleset. If the Ruleset is used for validation then the ruleName identifies the validation results of the various Rules of the Ruleset. The ruleName is optional and, if not specified, is assumed to be the progressive order number of the Rule in the Ruleset. However please note that, if ruleName is omitted, then the Rule names can change in case the Ruleset is modified, 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. In addition, if the results of more than one Ruleset have to be combined in one Data Set, then the user should make the relevant rulesetNames different.
codeItemRelation	specifies a (possibly conditioned) Code Item Relation. It expresses a logical relation between Code Items belonging to the Value Domain of the hrRulesetSignature, possibly conditioned by the Values of the Value Domains or Variables of the Conditioning Signature. The relation is expressed by one of the symbols **=**, **>**, **>=**, **<**, **<=**, that in this context denote special logical relationships typical of Code Items. The first member of the relation is a single Code Item. The second member of the relationship is the composition of one or more Code Items combined using the symbols **+** or **-**, which in turn also denote special logical operators typical of Code Items. The meaning of these symbols is better explained below and in the User Manual.
errorCode	a literal denoting the error code associated to the rule, to be assigned to the possible non-valid results in case the Rule is used for validation. If omitted then no error code is assigned (NULL value). VTL assumes that a Value Domain errorcode_vd of the error codes exists in the Information Model and contains all the possible error codes: the errorCode literal must be one of the possible Values of such a Value Domain. VTL assumes also that a Variable errorcode for describing the error codes exists in the IM and is a dependent variable of the Data Sets which contain the results of the validation.
errorLevel	a literal denoting the error level (severity) associated to the rule, to be assigned to the possible non-valid results in case the Rule is used for validation. If omitted then no error level is assigned (NULL value). VTL assumes that a Value Domain errorlevel_vd of the error levels exists in the Information Model and contains all the possible error levels: the errorLevel literal must be one of the possible Values of such a Value Domain. VTL assumes also that a Variable errorlevel for describing the error levels exists in the IM and is a dependent variable of the Data Sets which contain the results of the validation.
leftCondition	a *boolean* expression which defines the pre-condition for evaluating the left member Code Item (i.e., it is evaluated only when the leftCondition is TRUE); It can contain references to the Value domains or the Variables of the conditioningSignature of the Ruleset and Constants; all the VTL-ML component level operators are allowed. The

	leftCondition is optional, if missing it is assumed to be TRUE and the Rule is always evaluated.
leftCodeItem	a Code Item of the Value Domain specified in the hrRulesetSignature.
rightCodeItem	a Code Item of the Value Domain specified in the hrRulesetSignature.
rightCondition	a *boolean* scalar expression which defines the condition for a right member Code Item to contribute to the evaluation of the Rule (i.e., the right member Code Item is taken into account only when the relevant rightCondition is TRUE). It can contain references to the Value Domains or Variables of the vdConditioningSignature or varConditioningSignature of the Ruleset and Constants; all the VTL-ML component level operators are allowed. The rightCondition is optional, if omitted then it is assumed to be TRUE and the right member Code Item is always taken into account.

Input parameters type

rulesetName ::	name < ruleset >
ruleValueDomain ::	name <valuedomain >
condValueDomain ::	name <valuedomain >
vdAlias ::	name
ruleVariable ::	name
condVariable ::	name
varAlias ::	name
ruleName ::	name
errorCode ::	errorcode_vd
errorLevel ::	errorlevel_vd
leftCondition ::	boolean
leftCodeItem ::	name
rightCodeItem ::	name
rightCondition ::	boolean

Constraints

- leftCondition and rightCondition can refer only to Value Domains or Variables specified in vdConditioningSignature or varConditioningSignature.
- Either the ruleName is specified for all the Rules of the Ruleset or for none.
- If specified, the ruleName must be unique within the Ruleset.

Semantic specification

This operator defines a Hierarchical Ruleset named rulesetName that can be used both for validation and calculation purposes (see **check_hierarchy** and **hierarchy**). A Hierarchical Ruleset is a set of Rules expressing logical relationships between the Values (Code Items) of a Value Domain or a Represented Variable.
Each rule contains a Code Item Relation, possibly conditioned, which expresses the **relation between Code Items** to be enforced. In the relation, the left member Code Item is put in relation to a combination of one or more right member Code Items. The kinds of relations are described below.
The left member Code Item can be optionally conditioned through a leftCondition, a *boolean* expression which defines the cases in which the Rule has to be applied (if not declared the Rule is applied ever). The participation of each right member Code Item in the Relation can be optionally conditioned through a rightCondition, a *boolean* expression which defines the cases in which the Code Item participates in the relation (if not declared the Code Item participates to the relation ever).
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 that UnitedKingdom implies Europe (if a point belongs to United Kingdom it also belongs to Europe).

January2000 < year2000

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

means that January of the year 2000 implies the year 2000 (if a time instant belongs 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). 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 property 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), therefore the Rulesets of this kind are potentially largely reusable.

The Ruleset Signature specifies the space on which the Ruleset is defined, i.e., the ValueDomain or Variable on which the Code Item Relations are defined (the Ruleset is meant to be applicable to Data Sets having a Component which takes values on such a Value Domain or are defined by such a Variable). The optional vdConditioningSignature specifies the conditioning Value Domains (the conditions can refer only to those Value Domains), as well as the optional varConditioningSignature specifies the conditioning Variables (the conditions can refer only to those Variables).

The Hierarchical 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).

Within the Hierarchical Rulesets 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 in which the Rules are written in the Ruleset. The dependencies between the Rules form a directed acyclic graph.

The Hierarchical ruleset can be used for calculations to calculate the upper levels of the hierarchy if the data relevant to the leaves (or some other intermediate level) are available in the operand Data Set of the **hierarchy** operator (for more information see also the "Hierarchy" operator). 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.

When a Hierarchical Ruleset is used for calculation, only the Relations expressing coincidence (**=**) are evaluated (provided that the leftCondition is TRUE, and taking into account only right-side Code Items whose rightCondition 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), which are taken from the input Data Set (for more details about the evaluation options see the **hierarchy** operator). Moreover, the clauses typical of the validation are ignored (e.g., ErrorCode, ErrorLevel).

The Hierarchical Ruleset can be also used to filter the input Data Points. In fact 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 BeneluxRuleset ( valuedomain rule GeoArea) is
        Belgium = Belgium
      ; Luxembourg = Luxembourg
      ; Netherlands = Netherlands
```

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

The Hierarchical Rulesets can be used for validation in case various levels of detail are contained in the Data Set to be validated (see also the **check_hierarchy** operator for more details). The Hierarchical Rulesets express the coherency Rules between the different levels of detail. Because in the validation the various Rules can be evaluated independently, their order is not significant.
If a Hierarchical Ruleset is used for validation, all the possible Relations (**=**, **>**, **>=**, **<**, **<=**) are evaluated (provided that the leftCondition is TRUE and taking into account only right-side Code Items whose rightCondition is TRUE). The Rules are evaluated independently. Both the Code Items of the left and right members of the Relations are expected to belong to and taken from the input Data Set (for more details about the evaluation options see the **check_hierarchy** operator). The Antecedent Condition is evaluated and, if TRUE, 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 type of Relation. The possible relations in which Code Items are defined as equal to themselves are ignored. Further details are described in the **check_hierarchy** operator.
If the data to be validated are in different Data Sets, either they can be joined in advance using the proper VTL operators or the validation can be done by comparing those Data Sets directly, without using a Hierarchical Ruleset (see also the **check** operator).

Through the right and left Conditions, the Hierarchical Rulesets allow to declare the time validity of Rules and Relations. In fact leftCondition and RightCondition can be defined in term of the time Value Domain, expressing respectively when the left member Code Item has to be evaluated (i.e., when it is considered valid) and when a right member Code Item participates in the relation.
The following two simplified examples show possible ways of defining the European Union in term of participating Countries.

Example 1 (for simplicity the time literals are written without the needed "cast" operation)

```
define hierarchical ruleset  EuropeanUnionAreaCountries1
( valuedomain condition ReferenceTime as Time rule GeoArea ) is
        when between (Time, "1.1.1958", "31.12.1972")
            then EU = BE + FR + DE + IT + LU + NL
      ; when between (Time, "1.1.1973", "31.12.1980")
            then EU = … same as above … + DK + IE + GB
      ; when between (Time, "1.1.1981",  "02.10.1985")
            then EU = … same as above … + GR
      ; when between (Time, "1.1.1986",  "31.12.1994")
            then EU = … same as above … + ES + PT
      ; when between (Time, "1.1.1995",  "30.04.2004")
            then EU = … same as above … + AT + FI + SE
      ; when between (Time, "1.5.2004", "31.12.2006")
            then EU = … same as above … +CY+CZ+EE+HU+LT+LV+MT+PL+SI+SK
      ; when between (Time, "1.1.2007",  "30.06.2013")
            then EU = … same as above … + BG + RO
      ; when >= "1.7.2013"
            then EU = … same as above … + HR
end hierarchical ruleset
```

Example 2 (for simplicity the time literals are written without the needed "cast" operation)

```
define hierarchical ruleset  EuropeanUnionAreaCountries2
   (valuedomain condition ReferenceTime as Time rule GeoArea )  is
      EU =      AT  [ Time >= "0101.1995" ]
            + BE [ Time >= "01.01.1958" ]
            + BG [ Time >= "01.01.2007" ]

            +  …
            + SE [ Time >= "01.01.1995" ]
            + SI [ Time >= "01.05.2004" ]
            + SK [ Time >= "01.05.2004" ]
end hierarchical ruleset
```

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 rule Geo_Area) is
        World = Africa + America + Asia + Europe + Oceania
      ; Africa = Algeria + … + Zimbabwe
      ; America = Argentina + … + Venezuela
      ; Asia = Afghanistan + … + Yemen
      ; Europe = Albania + … + VaticanCity
      ; Oceania = Australia + … +  Vanuatu
      ; Afghanistan = AF_reg_01 + … + AF_reg_N
        … … … … … …
      ; Zimbabwe = ZW_reg_01 +  …  +  ZW_reg_M
      ; EuropeanUnion = … + … + … + …
      ; CentralAmericaCommonMarket = … + … + … + …
      ; OECD_Area = … + … + … + …
end hierarchical ruleset
```

The 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 RuleName is assigned to the different methods of breaking down the Transport.

```
define hierarchical ruleset  TransportBreakdown ( variable rule BoPItem )  is
        transport_method1 :    Transport  =  AirTransport + SeaTransport + LandTransport
      ; transport_method2 :    Transport = PassengersTransport + FreightsTransport
end hierarchical ruleset
```

Multiple relations can be useful even for 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 a homogeneous level of aggregation (e.g., by country), a possible Ruleset is the following:

```
define hierarchical ruleset  CalcCountryLevel ( valuedomain condition Residence rule GeoArea) is
        when Residence = “resident” then Country1  = Country1
      ; when Residence = “non-resident” then Country1 = Region11 + … + Region1M
        …
      ; when Residence = “resident” then CountryN = CountryN
      ; when Residence = “non-resident” then CountryN = Region N1 + … + RegionNM
end hierarchical ruleset
```

In the calculation, basically, for each Rule, for all the input Data Points and provided that the conditions are TRUE, the right Code Items are changed into the corresponding left Code Item, obtaining Data Points referred only to the left Code Items. Then the outcomes of all the Rules of the Ruleset are aggregated together to obtain the Data Points of the result Data Set.
As far as each left Code Item is calculated by means of a single Rule (i.e., a single calculation method), this process cannot generate inconsistencies.
Instead if a left Code Item is calculated by means of more Rules (e.g., through more than one calculation method), there is the risk of producing erroneous results (e.g., duplicated data), because the outcome of the multiple Rules producing the same Code Item are aggregated together. Proper definition of the left or right conditions can avoid this risk, ensuring that for each input Data Point just one Rule is applied.
If the Ruleset is aimed only at validation, there is no risk of producing erroneous results because in the validation the rules are applied independently.

Examples
1) The Hierarchical Ruleset is defined on the Value Domain “sex”: Total is defined as Male + Female.
No conditions are defined.

```
define hierarchical ruleset sex_hr  (valuedomain rule sex)  is
    TOTAL = MALE + FEMALE
end hierarchical ruleset
```

2) BENELUX is the aggregation of the Code Items BELGIUM, LUXEMBOURG and NETHERLANDS. No conditions are defined.

```
define hierarchical ruleset  BeneluxCountriesHierarchy (valuedomain rule GeoArea) is
    BENELUX = BELGIUM + LUXEMBOURG + NETHERLANDS errorcode “Bad value for Benelux”
end hierarchical ruleset
```

3) American economic partners. The first rule states that the value for North America should be 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 some elements. No conditions are defined.

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

4) Example of an aggregate Code Item having multiple definitions to be used for validation only. 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 validationruleset_bop (variable rule BoPItem ) is
        transport_method1 : Transport  =  AirTransport + SeaTransport + LandTransport
     ;  transport_method2 : Transport  =  PassengersTransport + FreightsTransport
end hierarchical ruleset
```

VTL-DL - User Defined Operators

define operator

Syntax

define operator operator_name **(** { parameter { **,** parameter }* } **)**
{ **returns** outputType }
is operatorBody
end define operator

parameter::= parameterName parameterType { **default** parameterDefaultValue }

Syntax description

operator_name	the name of the operator
parameter	the names of parameters, their data types and defaultvalues
outputType	the data type of the artefact returned by the operator
operatorBody	the expression which defines the operation
parameterName	the name of the parameter
parameterType	the data type of the parameter
parameterDefaultValue	the default value for the parameter (optional)

Parameters

operator_name	name
outputType	a VTL data type (see the Data Type Syntax below)
operatorBody	a VTL expression having the parameters (i.e., parameterName) as the operands
parameterName	name
parameterType	a VTL data type (see the Data Type Syntax below)
parameterDefaultValue	a Value of the same type as the parameter

Constraints

- Each parameterName must be unique within the list of parameters
- parameterDefaultValue must be of the same data type as the corresponding parameter
- outputType must be compatible with the type of operatorBody (it can also be a sub-type of the type returned by the operatorBody expression)
- If outputType is omitted then the type returned by the operatorBody expression is assumed
- If parameterDefaultValue is specified then the parameter is optional

Semantic specification

This operator defines a user-defined Operator by means of a VTL expression, specifying also the parameters, their data types, whether they are mandatory or optional and their (possible) default values.

Examples

Example1:

```
define operator max1 (x integer, y integer)
returns boolean is
if x > y then x else y
end define operator
```

Example2:

```
define operator add (x integer default 0, y integer default 0)
returns number is
x+y
end define operator
```

Data type syntax

The VTL data types are described in the VTL User Manual. Types are used throughout this Reference Manual as both meta-syntax and syntax.

They are used as meta-syntax in order to define the types of input and output parameters in the descriptions of VTL operators; they are used in the syntax, and thus are proper part of the VTL, in order to allow other operators to refer to specific data types. For example, when defining a custom operator (see the **define operator** above), one will need to declare the type of the input/output parameters.

The syntax of the data types is described below (as for the meaning of these definitions, see the section VTL Data Types in the User Manual). See also the section "Conventions for describing the operators' syntax" in the chapter "Overview of the language and conventions" above.

dataType ::= scalarType | compoundType

scalarType ::= { basicScalarType | valueDomainName | setName }^{1} { scalarTypeConstraint } { **null |** **not null** }

basicScalarType ::= { **scalar | number** | **integer** | **string** | **boolean** | **time** | **date** | **time_period** | **duration** }^{1}

valueDomainName :: name

setName :: name

scalarTypeConstraint ::= **[** valueBooleanCondition **]** | **{** scalarLiteral { **,** scalarLiteral }* **}**

compoundType ::= componentType | datasetType | operatorType | rulesetType | productType | universalSetType

componentType ::= componentRole { **<** scalar type **>** }

componentRole ::= { **component** | **identifier** | **measure** | **attribute** | **viral attribute** }^{1}

datasetType ::= **dataset {** componentConstraint { **,** componentConstraint }* **}**

componentConstraint ::= componentType { componentName | multiplicityModifier }^{1}

componentName :: name

multiplicityModifier ::= _ { **+** | ***** }

productType ::= { dataType { ***** dataType }$^{+}$ }^{1}

operatorType ::= { dataType **->** dataType }^{1}

rulesetType ::= { **ruleset** | dpRuleset | hrRuleset }^{1}

dpRuleset ::= **datapoint**
| **datapoint_on_valuedomains { (** name { ***** name }* **) }**
| **datapoint_on_variables { (** name { ***** name }* **) }**

hrRuleset ::= **hierarchical**
| **hierarchical_on_valuedomains {** valueDomainName { *
(prodValueDomains **)** } **}**
| **hierarchical_on_variables {** variableName { * **(** prodVariables **)** } **}**

universalSetType ::= **set** { **<** dataType **>** }

Note that the valueBooleanCondition in scalarTypeConstraint is expressed with reference to the fictitious variable "value" (see also the User Manual, section "Conventions for describing the Scalar Types"), which represents the generic value of the scalar type, for example:

integer { 0, 1 }	means an integer number whose value is 0 or 1
number [value >= 0]	means a number greater or equal than 0
string { "A", "B", "C" }	means a string whose value is A, B or C:

string [length (value) <= 10]   means a string whose length is lower or equal than 10:

General examples of the syntax for defining types can be found in the User Manual, section VTL Data Types and in the declaration of the data types of the VTL operators (sub-sections “input parameters type” and “result type”).

VTL-ML - Typical behaviours of the ML Operators

In this section, the common behaviours of some class of VTL-ML operators are described, both for a better understanding of the characteristics of such classes and to factor out and not repeat the explanation for each operator of the class.

Typical behaviour of most ML Operators

Unless differently specified in the Operator description, the Operators can be applied to Scalar Values, to Data Sets and to Data Set Components.

The operations on Scalar Values are primitive and are part of the core of the language. The other kind of operations can be typically be obtained by means of the scalar operations in conjunction with the Join operator, which is part of the core too.

In the operations on Data Set, the Operators are meant to be applied by default only to the values of the Measures of the input Data Sets, leaving the Identifiers unchanged. The Attributes follow by default their specific propagation rules, which are described in the User Manual.

In the operations on Components, the Operators are meant to be applied on the specified components of one input Data Set, in order to calculate a new component which becomes part of the resulting Data Set. In this case, the Attributes can be operated like the Measures.

Operators applicable on one Scalar Value or Data Set or Data Set Component

Operations on Scalar values

The operator is applied on a scalar value and returns a scalar value.

Operations on Data Sets

The operator is applied on a Data Set and returns a Data Set.

For example, using a functional style and denoting the operator with **f** (…), this can written as:

DS_r **:=** **f (** DS_1 **)**

The same operation, using an infix style and denoting the operator as **op**, can be also written as

DS_r **:=** **op** DS_1

This means that the operator is applied to the values of all the Measures of DS_1 in order to produce homonymous Measures in DS_r.

The application of the operator is allowed only if all the Measures of the operand Data Set are of a data type compatible with the operator (for example, a numeric operator is applicable only if all the Measures of the operand Data Sets are numeric). If the Measures of the operand Data Set are of different types, not all compatible with the operator to be applied, the membership or the keep clauses can be used to select only the proper Measures. No applicability constraints exist on Identifiers and Attributes, which can be any.

As for the data content, for each Data Point (DP_1) of the operand Data Set, a result Data Point (DP_r) is returned, having for the Identifiers the same values as DP_1.

For each Data Point DP_1 and for each Measure, the operator is applied on the Measure value of DP_1 and returns the corresponding Measure value of DP_r.

For each Data Point DP_1 and for each viral Attribute, the value of the Attribute propagates unchanged in DP_r.

As for the data structure, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes of the operand Data Set (DS_r maintains the Attributes declared as “viral” in DS_1; these Attributes are considered as “viral” also in DS_r, the “non-viral” Attributes of DS_1 are not kept in DS_r).

Operations on Data Set Components

The operator is applied on a Component (COMP_1) of a Data Set (DS_1) and returns another Component (COMP_r) which alters the structure of DS_1 in order to produce the result Data Set (DS_r).

For example, using a functional style and denoting the operator with f (…), this can be written as:

DS_r := DS_1 **[calc** COMP_r **:= f (** COMP_1**)]**

The same operation, using an infix style and denoting the operator as **op**, can be written as:

DS_r := DS_1 **[calc** COMP_r **:= op** COMP_1 **]**

This means that the operator is applied on COMP_1 in order to calculate COMP_r.

- If COMP_r is a new Component which originally did not exist in DS_1, it is added to the original Components of DS_1, by default as a Measure (unless otherwise specified), in order to produce DS_r.
- If COMP_r is one of the original Measures or Attributes of DS_1, the values obtained from the application of the operator f (…) replace the DS_1 original values for such a Measure or Attribute in order to produce DS_r.
- If COMP_r is one of the original Identifiers of DS_1, the operation is not allowed, because the result can become inconsistent.

In any case, an operation on the Components of a Data Set produces a new Data Set, as in the example above.

The application of the operator is allowed only if the input Component belongs to a data type compatible with the operator (for example, a numeric operator is applicable only on numeric Components). As already said, COMP_r cannot have the same name of an Identifier of DS_1.

As for the data content, for each Data Point DP_1 of DS_1, the operator is applied on the values of COMP_1 so returning the value of COMP_r.

As for the data structure, like for the operations on Data Sets above, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes of the operand Data Set (DS_r maintains the Attributes declared as "viral" in DS_1; these Attributes are considered as "viral" also in DS_r, the "non-viral" Attributes of DS_1 are not kept in DS_r). If an Attribute is explicitly calculated, the attribute propagation rule is overridden.

Moreover, in the case of the operations on Data Set Components, the (possible) new Component DS_r can be added to the original structure, the role of a (possible) existing DS_1 Component can be altered, the virality of a (possibly) existing DS_r Attribute can be altered, a (possible) COMP_r non-viral Attribute can be kept in the result. For the alteration of role and virality see also the **calc** clause.

Operators applicable on two Scalar Values or Data Sets or Data Set Components

Operation on Scalar values

The operator is applied on two Scalar values and returns a Scalar value.

Operation on Data Sets

The operator is applied either on two Data Sets or on one Data Set and one Scalar value and returns a Data Set. The composition of a Data Set and a Component is not allowed (it makes no sense).

For example, using a functional style and denoting the operator with **f** (…), this can be written as:

DS_r **:= f (** DS_1**,** DS_2 **)**

The same kind of operation, using an infix stile and denoting the operator as **op**, can be also written as

DS_r **:=** DS_1 **op** DS_2

This means that the operator is applied to the values of all the couples of Measures of DS_1 and DS_2 having the same names in order to produce homonymous Measures in DS_r. DS_1 or DS_2 may be replaced by a Scalar value.

The composition of two Data Sets (DS_1, DS_2) is allowed if the two operand Data Sets have exactly the same Measures and if all these Measures belong to a data type compatible with the operator (for example, a numeric operator is applicable only if all the Measures of the operand Data Sets are numeric). If the Measures of the operand Data Sets are different or of different types not all compatible with the operator to be applied, the membership or the **keep** clauses can be used to select only the proper Measures. The composition is allowed if these operand Data Sets have the same Identifiers or if one of them has at least all the Identifiers of the other one (in other words, the Identifiers of one of the Data Sets must be a superset of the Identifiers of the other one). No applicability constraints exist on the Attributes, which can be any.

As for the data content, the operand Data Sets (DS_1, DS_2) are joined to find the couples of Data Points (DP_1, DP_2), where DP_1 is from the first operand (DS_1) and DP_2 from the second operand (DS_2), which have the same values as for the common Identifiers. Data Points that are not coupled are left out (the inner join is used). An operand Scalar value is treated as a Data Point that couples with all the Data Points of the other operand. For each couple (DP_1, DP_2) a result Data Point (DP_r) is returned, having for the Identifiers the same values as DP_1 and DP_2.

For each Measure and for each couple (DP_1, DP_2), the Measure values of DP_1 and DP_2 are composed through the operator so returning the Measure value of DP_r. An operand Scalar value is composed with all the Measures of the other operand.

For each couple (DP_1, DP_2) and for each Attribute that propagates in DP_r, the Attribute value is calculated by applying the proper Attribute propagation algorithm on the values of the Attributes of DP_1 and DP_2 .

As for the data structure, the result Data Set (DS_r) has all the Identifiers (with no repetition of common Identifiers) and the Measures of both the operand Data Sets, and has the Attributes resulting from the application of the attribute propagation rules on the Attributes of the operands (DS_r maintains the Attributes declared as “viral” for the operand Data Sets; these Attributes are considered as “viral” also in DS_r, the “non-viral” Attributes of the operand Data Sets are not kept in DS_r).

Operation on Data Set Components

The operator is applied either on two Data Set Components (COMP_1, COMP_2) belonging to the same Data Set (DS_1) or on a Component and a Scalar value, and returns another Component (COMP_r) which alters the structure of DS_1 in order to produce the result Data Set (DS_r). The composition of a Data Set and a Component is not allowed (it makes no sense).

For example, using a functional style and denoting the operator with **f** (…), this can be written as:

DS_r := DS_1 **[calc** COMP_r **:= f (** COMP_1**,** COMP_2 **)]**

The same operation, using an infix style and denoting the operator as **op**, can be written as:

DS_r := DS_1 **[calc** COMP_r **:=** COMP_1 **op** COMP_2 **]**

This means that the operator is applied on COMP_1 and COMP_2 in order to calculate COMP_r.

- If COMP_r is a new Component which originally did not exist in DS_1, it is added to the original Components of DS_1, by default as a Measure (unless otherwise specified), in order to produce DS_r.
- If COMP_r is one of the original Measures or Attributes of DS_1, the values obtained from the application of the operator f (…) replace the DS_1 original values for such a Measure or Attribute in order to produce DS_r.
- If COMP_r is one of the original Identifiers of DS_1, the operation is not allowed, because the result can become inconsistent.

In any case, an operation on the Components of a Data Set produces a new Data Set, like in the example above.

The composition of two Data Set Components is allowed provided that they belong to the same Data Set[3]. Moreover, the input Components must belong to data types compatible with the operator (for example, a numeric operator is applicable only on numeric Components). As already said, COMP_r cannot have the same name of an Identifier of DS_1.

As for the data content, for each Data Point of DS_1, the values of COMP_1 and COMP_2 are composed through the operator so returning the value of COMP_r.

As for the data structure, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes of the operand Data Set (DS_r maintains the Attributes declared as “viral” in DS_1; these Attributes are considered as “viral” also in DS_r, the “non-viral” Attributes of DS_1 are not kept in DS_r). If an Attribute is explicitly calculated, the attribute propagation rule is overridden.

Moreover, in the case of the operations on Data Set Components, a (possible) new Component DS_r can be added to the original structure of DS_1, the role of a (possibly) existing DS_1 Component can be altered, the virality of a

[3] As obvious, the input Data Set can be the result of a previous composition of more other Data Sets, even within the same expression

(possibly) existing DS_r Attributes can be altered, a (possible) COMP_r non-viral Attribute can be kept in the result. For the alteration of role and virality see also the **calc** clause.

Operators applicable on more than two Scalar Values or Data Set Components

The cases in which an operator can be applied on more than two Data Sets (like the Join operators) are described in the relevant sections.

Operation on Scalar values

The operator is applied on more Scalar values and returns a Scalar value according to its semantics.

Operation on Data Set Components

The operator is applied either on a combination of more than two Data Set Components (COMP_1, COMP_2) belonging to the same Data Set (DS_1) or Scalar values, and returns another Component (COMP_r) which alters the structure of DS_1 in order to produce the result Data Set (DS_r). The composition of a Data Set and a Component is not allowed (it makes no sense).

For example, using a functional style and denoting the operator with **f** (…), this can be written as:

DS_r := DS_1 **[substr** COMP_r **:= f (** COMP_1**,** COMP_2, COMP_3 **)]**

This means that the operator is applied on COMP_1, COMP_2 and COMP_3 in order to calculate COMP_r.

- If COMP_r is a new Component which originally did not exist in DS_1, it is added to the original Components of DS_1, by default as a Measure (unless otherwise specified), in order to produce DS_r.
- If COMP_r is one of the original Measures or Attributes of DS_1, the values obtained from the application of the operator f (…) replace the DS_1 original values for such a Measure or Attribute in order to produce DS_r.
- If COMP_r is one of the original Identifiers of DS_1, the operation is not allowed, because the result can become inconsistent.

In any case, an operation on the Components of a Data Set produces a new Data Set, like in the example above.

The composition of more Data Set Components is allowed provided that they belong to the same Data Set[4]. Moreover, the input Components must belong to data types compatible with the operator (for example, a numeric operator is applicable only on numeric Components). As already said, COMP_r cannot have the same name of an Identifier of DS_1.

As for the data content, for each Data Point of DS_1, the values of COMP_1, COMP_2 and COMP_3 are composed through the operator so returning the value of COMP_r.

As for the data structure, the result Data Set (DS_r) has the Identifiers and the Measures of the operand Data Set (DS_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes of the operand Data Set (DS_r maintains the Attributes declared as "viral" in DS_1; these Attributes are considered as "viral" also in DS_r, the "non-viral" Attributes of DS_1 are not kept in DS_r). If an Attribute is explicitly calculated, the attribute propagation rule is overridden.

Moreover, in the case of the operations on Data Set Components, a (possible) new Component DS_r can be added to the original structure of DS_1, the role of a (possibly) existing DS_1 Component can be altered, the virality of a (possibly) existing DS_r Attributes can be altered, a (possible) COMP_r non-viral Attribute can be kept in the result. For the alteration of role and virality see also the **calc** clause.

Behaviour of Boolean operators

The Boolean operators are allowed only on operand Data Sets that have a single measure of type *boolean*. As for the other aspects, the behaviour is the same as the operators applicable on one or two Data Sets described above.

[4] As obvious, the input Data Set can be the result of a previous composition of more other Data Sets, even within the same expression

Behaviour of Set operators

These operators apply the classical set operations (union, intersection, difference, symmetric differences) to the Data Sets, considering them as sets of Data Points. These operations are possible only if the Data Sets to be operated have the same data structure, and therefore the same Identifiers, Measures and Attributes[5].

Behaviour of Time operators

The *time* operators are the operators dealing with *time*, *date* and *time_period* basic scalar types. These types are described in the User Manual in the sections "Basic Scalar Types" and "External representations and literals used in the VTL Manuals".

The time-related formats used for explaining the time operators are the following (they are described also in the User Manual).

For the *time* values:

YYYY-MM-DD/YYYY-MM-DD

Where *YYYY* are 4 digits for the year, *MM* two digits for the month, *DD* two digits for the day. For example:

2000-01-01/2000-12-31	the whole year 2000
2000-01-01/2009-12-31	the first decade of the XXI century

For the *date* values:

YYYY-MM-DD

The meaning of the symbols is the same as above. For example:

2000-12-31	the 31st December of the year 2000
2010-01-01	the first of January of the year 2010

For the *time_period* values:

YYYY{P}{NNN}

Where *YYYY* are 4 digits for the year, *P* is one character for the period indicator of the regular period (it refers to the *duration* data type and can assume one of the possible values listed below), *NNN* are from zero to three digits which contain the progressive number of the period in the year. For annual data the A and the three digits NNN can be omitted. For example:

2000M12	the month of December of the year 2000 (duration: M)
2010Q1	the first quarter of the year 2010 (duration: Q)
2010A	the whole year 2010 (duration: A)
2010	the whole year 2010 (duration: A)

For the *duration* values, which are the possible values of the period indicator of the regular periods above, it is used for simplicity just one character whose possible values are the following:

Code	*Duration*
D	Day
W	Week
M	Month
Q	Quarter
S	Semester
A	Year

As mentioned in the User Manual, these are only examples of possible time-related representations, each VTL system is free of adopting different ones. In fact no predefined representations are prescribed, VTL systems are free to using they preferred or already existing ones.

Several time operators deal with the specific case of Data Sets of time series, having an Identifier component that acts as the reference time and can be of one of the scalar types *time*, *date* or *time_period*; moreover this Identifier must be periodical, i.e. its possible values are regularly spaced and therefore have constant duration (frequency).

[5] According to the VTL IM, the Variables that have the same name have also the same data type

It is worthwhile to recall here that, in the case of Data Sets of time series, VTL assumes that the information about which is the Identifier Components that acts as the reference time and which is the period (frequency) of the time series exists and is available in some way in the VTL system. The VTL Operators are aware of which is the reference time and the period (frequency) of the time series and use these information to perform correct operations. VTL also assumes that a Value Domain representing the possible periods (e.g. the period indicator Value Domain shown above) exists and refers to the *duration* scalar type. For the assumptions above, the users do not need to specify which is the Identifier Component having the role of reference time.

The operators for time series can be applied only on Data Sets of time series and returns a Data Set of time series. The result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set and contains the same time series as the operand. The Attribute propagation rule is not applied.

Operators changing the data type

These Operators change the Scalar data type of the operands they are applied to (i.e. the type of the result is different from the type of the operand). For example, the **length** operator is applied to a value of *string* type and returns a value of *integer* type. Another example is the **cast** operator.

Operation on Scalar values

The operator is applied on (one or more) Scalar values and returns one Scalar value of a different data type.

Operation on Data Sets

If an Operator change the data type of the Variable it is applied to (e.g., from *string* to *number*), the result Data Set cannot maintain this Variable as it happens in the previous cases, because a Variable cannot have different data types in different Data Sets[6].

As a consequence, the converted variable cannot follow the same rules described in the sections above and must be replaced, in the result Data Set, by another Variable of the proper data type.

For sake of simplicity, the operators changing the data type are allowed only on mono-measure operand Data Sets, so that the conversion happens on just one Measure. A default generic Measure is assigned by default to the result Data Set, depending on the data type of the result (the default Measure Variables are reported in the table below).

Therefore, if the operands are originally multi-measure, just one Measure must be pre-emptively selected (for example through the membership operator) in order to apply the changing-type operator. Moreover, if in the result Data Set a different Measure Variable name is desired than the one assigned by default, it is possible to change the Variable name (see the **rename** operator).

As for the Identifiers and the Attributes, the behaviour of these operators is the same as the typical behaviour of the unary or binary operators.

Operation on Data Set Components

For the same reasons above, the result Component cannot be the same as one of the operand Components and must be of the appropriate Scalar data type.

Default Names for Variables and Value Domains used in this manual

The following table shows the default Variable names and the relevant default Value Domain. These are only the names used in this manual for explanatory purposes and can be personalised in the implementations. If VTL rules are exchanged, the personalised names need to be shared with the partners of the exchange.

Scalar data type	Default Variable	Default Value Domain
string	string_var	string_vd

[6] This according both to the mathematical meaning of a Variable and the VTL Information Model; in fact a Represented Variable is defined on just one Value Domain, which has just one data type, independently of the Data Structures and the Data Sets in which the Variable is used.

number	num_var	num_vd
integer	int_var	int_vd
time	time_var	time_vd
time_period	time_period_var	time_period_vd
date	date_var	date_vd
duration	duration_var	duration_vd
boolean	bool_var	bool_vd

Type Conversion and Formatting Mask

The conversions between *scalar* types is provided by the operator **cast**, described in the section of the general purpose operators. Some particular types of conversion require the specification of a formatting mask, which specifies which format the source or the destination of the conversion should assume. The formatting masks for the various scalar types are explained here.

If needed, the formatting Masks can be personalized in the VTL implementations. If VTL rules are exchanged, the personalised masks need to be shared with the partners of the exchange.

The Numbers Formatting Mask

The ***number* formatting mask** can be defined as a combination of characters whose meaning is the following:

- "D" one numeric digit (if the scientific notation is adopted, D is only for the mantissa)
- "E" one numeric digit (for the exponent of the scientific notation)
- "*" an arbitrary number of digits
- "+" at least one digit
- "." (dot) can be used as a separator between the integer and the decimal parts.
- "," (comma) can be used as a separator between the integer and the decimal parts.

Examples of valid masks are:

DD.DDDDD, DD.D, D, D.DDDD, D*.D*, D+.D+ , DD.DDDEEEE

The Time Formatting Mask

The format of the values of the types *time*, *date* and *time_period* can be specified through specific formatting masks. A mask related to *time*, *date* and *time_period* is formed by a sequence of symbols which denote:

- the time units that are used, for example years, months, days
- the format in which they are represented, for example 4 digits for the year (2018), 2 digits for the month within the year (04 for April) and 2 digits for the day within the year and the month (05 for the 5th)
- the order of these parts; for example, first the 4 digits for the year, then the 2 digits for the month and finally the 2 digits for the day
- other (possible) typographical characters used in the representation; for example, a line between the year and the month and between the month and the day (e.g., 2018-04-05).

The time formatting masks follows the general rules below.

For a numerical representations of the time units:

- A digit is denoted through the use of a **special character** which depends on the time unit. for example Y is for "year", M is for "month" and D is for "day"
- The special character is lowercase for the time units shorter than the day (for example h for "hour", m for "minute", s for "second") and uppercase for time units equal to "day" or longer (for example W for "week", Q for "quarter", S for "semester")

- The number of letters matches the number of digits, for example YYYY means that the year is represented with four digits and MM that the month is of 2 digits
- The numerical representation is assumed to be padded by leading 0 by default, for example MM means that April is represented as 04 and the year 33 AD as 0033
- If the numerical representation is not padded, the optional digits that can be omitted (if equal to zero) are enclosed within braces; for example {M}M means that April is represented by 4 and December by 12, while {YYY}Y means that the 33 AD is represented by 33

For textual representations of the time units:

- **Special words** denote a textual localized representation of a certain unit, for example DAY means a textual representation of the day (MONDAY, TUESDAY ...)
- An optional number following the special word denote the maximum length, for example DAY3 is a textual representation that uses three characters (MON, TUE ...)
- The case of the special word correspond to the case of the value; for example day3 (lowercase) denotes the values mon, tue ...
- The case of the initial character of the special word correspond to the case of the initial character of the time format; for example Day3 denotes the values Mon, Tue ...
- The letter P denotes the period indicator, (i.e., day, week, month ...) and the letter p denotes ond digit for the number of periods

Representation of more time units:

- If more time units are used in the same mask (for example years, months, days), it is assumed that the more detailed units (e.g., the day) are expressed through the order number that they assume within the less detailed ones (e.g., the month and the year). For example, if years, weeks and days are used, the weeks are within the year (from 1 to 53) and the days are within the year and the week (from 1 to 7).
- The position of the digits in the mask denotes the position of the corresponding values; for example, YYYMMDD means four digits for the year followed by two digits for the month and then two digits for the day (e.g., 20180405 means the year 2018, month April, day 5^{th})
- Any other character can be used in the mask, meaning simply that it appears in the same position; for example, YYYY-MM-DD means that the values of year, month and day are separated by a line (e.g., 2018-04-05 means the year 2018, month April, day 5^{th}) and \PMM denotes the letter “P” followed by two characters for the month.
- The special characters and the special words, if prefixed by the reverse slash (\) in the mask, appear in the same position in the time format; for example \PMM\M means the letter “P” followed by two characters for the month and then the letter “M”; for example, P03M means a period of three months (this is an ISO 8601 standard representation for a period of MM months). The reverse slash can appear in the format if needed by prefixing it with another reverse slash; for example YYYY\\MM means for digits for the year, a reverse slash and two digits for the month.
-

The **special characters** and the corresponding time units are the following:

C	century
Y	year
S	semester
Q	quarter
M	month
W	week
D	day
h	hour digit (by default on 24 hours)
m	minute
s	second
d	decimal of second
P	period indicator (see the “duration” codes below)
p	number of periods

The **special words** for textual representations are the following:

AM/PM indicator of AM / PM (e.g. am/pm for "am" or "pm")
MONTH textual representation of the month (e.g., JANUARY for January)
DAY textual representation of the day (e.g., MONDAY for Monday)

Examples of formatting masks for the *time* scalar type:

A Scalar Value of type *time* denotes time intervals of any duration and expressed with any precision, which are the intervening time between two time points.

These examples are about three possible ISO 8601 formats for expressing time intervals:

- Start and end time points, such as "2015-03-03T09:30:45Z/2018-04-05T12:30:15Z"
  VTL Mask: YYYY-MM-DDThh:mm:ssZ/YYYY-MM-DDThh:mm:ssZ
- Start and duration, such as "2015-03-03T09:30:45-01/P1Y2M10DT2H30M"
  VTL Mask: YYYY-MM-DDThh:mm:ss-01/PY\YM\MDD\DT{h}h\Hmm\M
- Duration and end, such as "P1Y2M10DT2H30M/2018-04-05T12:30:00+02"
  VTL Mask: PY\YM\MDD\DT{h}h\Hmm\M/YYYY-MM-DDThh:mm:ssZ

Example of other possible ISO formats having accuracy reduced to the day

- Start and end, such as "20150303/20180405"
  VTL Mask: YYYY-MM-DD/YYYY-MM-DD
- Start and duration, such as "2015-03-03/P1Y2M10D"
  VTL Mask: YYYY-MM-DD/PY\YM\MDD\D
- Duration and end, such as "P1Y2M10D/2018-04-05"
  VTL Mask: PY\YM\MDD\DT/YYYY-MM-DD

Examples of formatting masks for the *date* scalar type:

A *date* scalar type is a point in time, equivalent to an interval of time having coincident start and end duration equal to zero.

These examples about possible ISO 8601 formats for expressing dates:

- Date and day time with separators: "2015-03-03T09:30:45Z"
  VTL Mask: YYYY-MM-DDThh:mm:ssZ
- Date and day time without separators "20150303T093045-01 "
  VTL Mask: YYYYMMDDThhmmss-01

Example of other possible ISO formats having accuracy reduced to the day

- Date and day-time with separators "2015-03-03/2018-04-05"
  VTL Mask: YYYY-MM-DD/YYYY-MM-DD
- Start and duration, such as "2015-03-03/P1Y2M10D"
  VTL Mask: YYYY-MM-DD/PY\YM\MDD\D

Examples of formatting masks for the *time_period* scalar type:

A *time_period* denotes non-overlapping time intervals having a regular duration (for example the years, the quarters of years, the months, the weeks and so on). The time_period values include the representation of the duration of the period.

These examples are about possible formats for expressing time-periods:

- Generic time period within the year such as: "2015Q4", "2015M12""2015D365"
  VTL Mask: YYYYP{ppp} where P is the period indicator and ppp three digits for the number of periods, in the values, the period indicator may assume one of the values of the duration scalar type listed below.
- Monthly period: "2015M03"
  VTL Mask: YYYY\MMM

Examples of formatting masks for the *duration* scalar type:

A Scalar Value of type *duration* denotes the length of a time interval expressed with any precision and without connection to any particular time point (for example one year, half month, one hour and fifteen minutes).

These examples are about possible formats for expressing durations (period / frequency)

- Non ISO representation of the *duration* in one character, whose possible codes are:

Code	*Duration*
D	Day
W	Week
M	Month
Q	Quarter
S	Semester
A	Year

  VTL Mask: P (period indicator)

- ISO 8601 composite duration: "P10Y2M12DT02H30M15S" (P stands for "period")

  VTL Mask: \PYY\YM\MDD\DThh\Hmm\Mss\S

- ISO 8601 duration in weeks: "P018W" (P stands for "period")

  VTL Mask: \PWWW\W

- ISO 4 characters representation: P10M (ten months), P02Q (two quarters) …

  VTL Mask: \PppP

Examples of fixed characters used in the ISO 8601 standard which can appear as fixed characters in the relevant masks:

P	designator of duration
T	designator of time
Z	designator of UTC zone
"+"	designator of offset from UTC zone
"-"	designator of offset form UTC zone
/	time interval separator

Attribute propagation

The VTL has different default behaviours for Attributes and for Measures, to comply as much as possible with the relevant manipulation needs. At the Data Set level, the VTL Operators manipulate by default only the Measures and not the Attributes. At the Component level, instead, Attributes are calculated like Measures, therefore the algorithms for calculating Attributes, if any, can be specified explicitly in the invocation of the Operators. This is the behaviour of clauses like **calc**, **keep**, **drop**, **rename** and so on, either inside or outside the join (see the detailed description of these operators in the Reference Manual).

The users which want to automatize the propagation of the Attributes' Values can optionally enforce a mechanism, called Attribute Propagation rule, whose behaviour is explained in the User Manual (see the section "Behaviour for Attribute Components"). The adoption of this mechanism is optional, users are free to allow the attribute propagation rule or not. The users that do not want to allow Attribute propagation rules simply will not implement what follows.

In short, the automatic propagation of an Attribute depends on a Boolean characteristic, called "virality", which can be assigned to any Attribute of a Data Set (a viral Attribute has virality = TRUE, a non-viral Attribute has virality=FALSE, if the virality is not defined, the Attribute is considered as non-viral).

By default, an Attribute propagates from the operand Data Sets (DS_i) to the result Data Set (DS_r) if it is "viral" at least in one of the operand Data Sets. By default, an Attribute which is viral in one of the operands DS_i is considered as viral also in the result DS_r.

The Attribute propagation rule does not apply for the time series operators.

The Attribute propagation rule does not apply if the operations on the Attributes to be propagated are explicitly specified in the expression (for example through the **keep** and **calc** operators). This way it is possible to keep in the result also Attribute which are non-viral in all the operands, to drop viral Attributes, to override the (possible) default calculation algorithm of the Attribute, to change the virality of the resulting Attributes.

VTL-ML - General purpose operators

Parentheses : ( )

Syntax

(op **)**

Input parameters

op the operand to be evaluated before performing other operations written outside the parentheses. According to the general VTL rule, operators can be nested, therefore any Data Set, Component or scalar op can be obtained through an expression as complex as needed (for example op can be written as the expression 2 + 3).

Examples of valid syntaxes

```
( DS_1 + DS_2 )
( CMP_1 - CMP_2 )
( 2 + DS_1 )
( DS_2 - 3 * DS_3 )
```

Semantic for scalar operations

Parentheses override the default evaluation order of the operators that are described in the section "VTL-ML – Evaluation order of the Operators". The operations enclosed in the parentheses are evaluated first. For example (2+3)*4 returns 20, instead 2+3*4 returns 14 because the multiplication has higher precedence than the addition.

Input parameters type

op :: dataset
| component
| scalar

Result type

result :: dataset
| component
| scalar

Additional constraints

None.

Behaviour

As mentioned, the op of the parentheses can be obtained through an expression as complex as needed (for example op can be written as DS_1 - DS_2. The part of the expression inside the parentheses is evaluated before the part outside of the parentheses. If more parentheses are nested, the inner parentheses are evaluated first, for example (20 – 10 / (2 + 3)) * 3 would give 54.

Examples

```
(DS_1 + DS_2) * DS_3
(CMP_1 – CMP_2 / (CMP_3 + CMP_4) ) * CMP_5
```

Persistent assignment : <-

Syntax

re **<-** op

Input Parameters

re the result

op the operand. According to the general VTL rule allowing the indentation of the operators, op can be obtained through an expression as complex as needed (for example op can be the expression DS_1 - DS_2).

Examples of valid syntaxes

```
DS_r <- DS_1
DS_r <- DS_1 - DS_2
```

Semantics for scalar operations

empty

Input parameters type

op :: dataset

Result type

result :: dataset

Additional constraints

The assignment cannot be used at Component level because the result of a Transformation cannot be a Data Set Component. When operations at Component level are invoked, the result is the Data Set which the output Components belongs to.

Behaviour

The input operand op is assigned to the **persistent** result re, which assumes the same value as op. As mentioned, the operand op can be obtained through an expression as complex as needed (for example op can be the expression DS_1 - DS_2).

The result re is a persistent Data Set that has the same data structure as the Operand. For example in DS_r <- DS_1 the data structure of DS_r is the same as the one of DS_1.

If the Operand op is a scalar value, the result Data Set has no Components and contains only such a scalar value. For example, income <- 3 assigns the value 3 to the persistent Data Set named income.

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
2013	Belgium	5	5
2013	Denmark	2	10
2013	France	3	12
2013	Spain	4	20

Example 1: DS_r <- DS_1 results in:

DS_r (persistent Data Set)			
Id_1	**Id_2**	**Me_1**	**Me_2**
2013	Belgium	5	5
2013	Denmark	2	10
2013	France	3	12
2013	Spain	4	20

Non-persistent assignment : :=

Syntax

re := op

Input parameters

re the result

op the operand (according to the general VTL rule allowing the indentation of the operators, op can be obtained through an expression as complex as needed (for example op can be the expression DS_1 - DS_2).

Examples of valid syntaxes

DS_r := DS_1
DS_r := 3
DS_r := DS_1 - DS_2
DS_r := 3 + 2

Semantic for scalar operations

empty

Input parameters type

op :: dataset
| scalar

Result type

result :: dataset

Additional constraints

The assignment cannot be used at Component level because the result of a Transformation cannot be a Data Set Component. When operations at Component level are invoked, the result is the Data Set which the output Components belongs to.
The same symbol denoting the non-persistent assignment Operator (**:=**) is also used inside other operations at Component level (for example in **calc** and **aggr**) in order to assign the result of the operation to the output Component: please note that in these cases the symbol **:=** does not denote the non-persistent assignment (i.e., this Operator), which cannot operate at Component level, but a special keyword of the syntax of the other Operator in which it is used.

Behaviour

The value of the operand op is assigned to the result re, which is non-persistent and therefore is not stored. As mentioned, the operand op can be obtained through an expression as complex as needed (for example op can be the expression DS_1 - DS_2).
The result re is a non-persistent Data Set that has the same data structure as the Operand. For example in DS_r := DS_1 the data structure of DS_r is the same as the one of DS_1.
If the Operand op is a scalar value, the result Data Set has no Components and contains only such a scalar value. For example, income := 3 assigns the value 3 to the non-persistent Data Set named income.

Examples

Given the operand Data Sets DS_1:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
2013	Belgium	5	5
2013	Denmark	2	10
2013	France	3	12
2013	Spain	4	20

Example 1: DS_r := DS_1 results in:

DS_r (non persistent Data Set)			
Id_1	**Id_2**	**Me_1**	**Me_2**
2013	Belgium	5	5
2013	Denmark	2	10
2013	France	3	12
2013	Spain	4	20

Membership :

Syntax

ds#comp

Input Parameters

ds the Data Set
comp the Data Set Component

Examples of valid syntaxes

DS_1#COMP_3

Semantic for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

ds :: dataset
comp :: name < component >

Result type

result :: dataset

Additional constraints

comp must be a Data Set Component of the Data Set ds

Behaviour

The membership operator returns a Data Set having the same Identifier Components of ds and a single Measure. If comp is a Measure in ds, then comp is maintained in the result while all other Measures are dropped. If comp is an Identifier or an Attribute Component in ds, then all the existing Measures of ds are dropped in the result and a new Measure is added. The Data Points' values for the new Measure are the same as the values of comp in ds. A default conventional name is assigned to the new Measure depending on its type: for example num_var if the Measure is *numeric*, string_var if it is *string* and so on (the default name can be renamed through the **rename** operator if needed).
The Attributes follow the Attribute propagation rule as usual (viral Attributes of ds are maintained in the result as viral, non-viral ones are dropped). If comp is an Attribute, it follows the Attribute propagation rule too.
The same symbol denoting the membership operator (#) is also used inside other operations at Component level (for example in **join**, **calc**, **aggr**) in order to identify the Components to be operated: please note that in these cases the symbol # does not denote the membership operator (i.e., this operator, which does not operate at Component level), but a special keyword of the syntax of the other operator in which it is used.

Examples

Given the operand Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Me_1**	**Me_2**	**At_1**
1	A	1	5	
1	B	2	10	P
2	A	3	12	

Example 1: DS_r := DS_1#Me_1 results in:

(assuming that At_1 is not viral in DS_1)

DS_r		
Id_1	**Id_2**	**Me_1**
1	A	1
1	B	2
2	A	3

(assuming that At_1 is viral in DS_1)

DS_r			
Id_1	**Id_2**	**Me_1**	**At_1**
1	A	1	
1	B	2	P
2	A	3	

Example 2: DS_r := DS_1#Id_1 assuming that At_1 is viral in DS_1 results in:

DS_r			
Id_1	**Id_2**	**num_var**	**At_1**
1	A	1	
1	B	1	P
2	A	2	

Example 3: DS_r := DS_1#At_1 assuming that At_1 is viral in DS_1 results in:

DS_r			
Id_1	**Id_2**	**string_var**	**At_1**
1	A		
1	B	P	P
2	A		

User-defined operator call

Syntax

operatorName **(** { argument { , argument }* } **)**

Input parameters

operatorName	the name of an existing user-defined operator
argument	argument passed to the operator

Examples of valid syntaxes

max1 (2, 3)

Semantic for scalar operations

It depends on the specific user-defined operator that is invoked.

Input parameters type

operatorName ::	name
argument ::	A data type compatible with the type of the parameter of the user-defined operator that is invoked (see also the "Type syntax" section).

Result type

result ::	The data type of the result of the user-defined operator that is invoked (see also the "Type syntax" section).

Additional constraints

- operatorName must refer to an operator created with the **define operator** statement.
- The type of each argument value must be compliant with the type of the corresponding parameter of the user defined operator (the correspondence is in the positional order).

Behaviour

The invoked user-defined operator is evaluated. The arguments passed to the operator in the invocation are associated to the corresponding parameters in positional order, the first argument as the value of the first parameter, the second argument as the value of the second parameter, and so on. An underscore ("_") can be used to denote that the value for an optional operand is omitted. One or more optional operands in the last positions can be simply omitted.

Examples

Example 1:

Definition of the max1 operator (see also "define operator" in the VTL-DL):

```
define operator max1 (x integer, y integer)
returns boolean
is   if x > y then x else y
end define operator
```

User-defined operator call of the max1 operator:

```
max1 ( 2, 3 )
```

Evaluation of an external routine : eval

Syntax

eval **(** externalRoutineName **(** { argument } { **,** argument }* **),** language**,** **returns** outputType **)**

Input parameters

externalRoutineName	the name of an external routine
argument	the arguments passed to the external routine
language	the implementation language of the routine
outputType	the data type of the object returned by eval (see the section: Data type syntax)

Examples of valid syntaxes

eval (routine1 (DS_1))

Semantics for scalar operations:

This is not a scalar operation.

Input parameters type

externalRoutineName ::	name
argument ::	any data type
language ::	string
outputType ::	any data type restricting Data Set or scalar

Result Type

result ::	dataset

Additional constraints

- The **eval** is the only VTL Operator that <u>does not allow nesting</u> and therefore a Transformation can contain just one invocation of **eval** and no other invocations. In other words, **eval** cannot be nested as the operand of another operation as well as another operator cannot be nested as an operand of **eval**
- The result of an expression containing **eval** must be persistent
- externalRoutineName is the conventional name of a non-VTL routine
- the invoked external routine must be consistent with the VTL principles, first of all its behaviour must be functional, so having in input and providing in output first-order functions
- argument is an argument passed to the external routine, it can be a name or a value of a VTL artefacts or some other parameter required by the routine
- the arguments passed to the routine correspond to the parameters of the invoked external routine in positional order; as usual the optional parameters are substituted by the underscore if missing. The conversion of the VTL input/output data types from and to the external routine processor is left to the implementation.

Behaviour

The **eval** operator invokes an external, non-VTL routine, and returns its result as a Data Set or a scalar. The specific data type can be given in the invocation. The routine specified in the **eval** operator can perform any internal logic.

Examples

Assuming that SQL3 is an SQL statement which produces DS_r starting from DS_1:

```
DS_r := eval( SQL3( DS_1 ) , "SQL",
              returns dataset { identifier<geo_area> ref_area,
                                identifier<date> time,
                                measure<number> obs_value,
                                attribute<string> obs_status }  )
```

Assuming that f is an externally defined Java method:

```
DS_r := DS_1[calc Me := eval( f(Me) + 1, "Java", integer) ]
```

Type conversion :   cast

Syntax

cast (op , scalarType { **,** mask} **)**

Input parameters

op	the operand to be cast
scalarType	the name of the scalar type into which op has to be converted
mask	a character literal that specifies the format of op

Examples of valid syntaxes

See the examples below.

Semantics for scalar operations:

This operator converts the scalar type of op to the scalar type specified by scalarType. It returns a copy of op converted to the specified scalarType.

Input parameters type

op :: dataset{ measure<scalar> _ }
| component<scalar>
| scalar
scalarType :: scalar type (see the section: Data type syntax)
mask :: string

Result type

result :: dataset{ measure<scalar> _ }
| component<scalar>
| scalar

Additional constraints

- Not all the conversions are possible, the specified casting operation is allowed only according to the semantics described below.
- The mask must adhere to one of the formats specified below.

Behaviour

Conversions between basic scalar types

The VTL assumes that a basic scalar type has a unique internal and more possible external representations (formats).

The external representations are those of the Value Domains which refers to such a basic scalar types (more Value Domains can refer to the same basic scalar type, see the VTL Data Types in the User Manual). For example, there can exist a *boolean* Value Domain which uses the values TRUE and FALSE and another *boolean* Value Domain which uses the values 1 and 0. The external representations are the ones of the Data Point Values and are obviously known by users.

The unique internal representation of a basic scalar type, instead, is used by the **cast** operator as a technical expedient to make the conversion between external representations easier: not necessarily users are aware of it. In a conversion, the **cast** converts the source external representation into the internal representation (of the corresponding scalar type), then this last one is converted into the target external representation (of the target type). As mentioned in the User Manual, VTL does not prescribe any specific internal representation for the various scalar types, leaving different organisations free of using their preferred or already existing ones.

In some cases, depending on the type of op, the output scalarType and the invoked operator, an automatic conversion is made, that is, even without the explicit invocation of the **cast** operator: this kind of conversion is called **implicit casting**.

In other cases, more than all when the implicit casting is not possible, the type conversion must be specified explicitly through the invocation of the **cast** operator: this kind of conversion is called **explicit casting**. If an explicit casting is specified, the (possible) implicit casting is overridden. There are two main categories of implicit casting:

- "**Explicit with mask**": the explicit conversion requires a formatting mask that specifies how the actual casting is performed;
- "**Explicit w/o mask**": the explicit conversion does not requires a formatting mask.

The table below summarises the possible castings between the basic scalar types. In particular, the input type is specified in the first column (row headings) and the output type in the first row (column headings).

Expected→ *Provided*	***integer***	***number***	***boolean***	***time***	***date***	***time_period***	***string***	***duration***
integer	-	*Implicit*	*Explicit w/o mask*	*Not feasible*	*Not feasible*	*Not feasible*	*Implicit*	*Not feasible*

number	*Explicit w/o mask*	-	*Explicit w/o mask*	*Not feasible*	*Not feasible*	*Not feasible*	*Implicit*	*Not feasible*
boolean	*Explicit w/o mask*	*Explicit w/o mask*	-	*Not feasible*	*Not feasible*	*Not feasible*	*Implicit*	*Not feasible*
time	*Not feasible*	*Not feasible*	*Not feasible*	-	*Not feasible*	*Not feasible*	*Explicit with mask*	*Not feasible*
date	*Not feasible*	*Not feasible*	*Not feasible*	*Implicit*	-	*Explicit w/o mask*	*Explicit with mask*	*Not feasible*
time_period	*Not feasible*	*Not feasible*	*Not feasible*	*Implicit*	*Explicit with mask*	-	*Explicit w/o mask*	*Not feasible*
string	*Explicit w/o mask*	*Explicit with mask*	*Not feasible*	*Explicit with mask*	*Explicit with mask*	*Explicit with mask*	-	*Explicit with mask*
duration	*Not feasible*	*Not feasible*	*Not feasible*	*Not feasible*	*Not feasible*	*Not feasible*	*Explicit with mask*	-

The type of casting can be personalised in specific environments, provided that the personalisation is explicitly documented with reference to the table above. For example, assuming that an explicit **cast** with mask is required and that in a specific environment a definite mask is used for such a kind of conversions, the **cast** can also become implicit provided that the mask that will be applied is specified.

The **implicit casting** is performed when a value of a certain type is provided when another type is expected. Its behaviour is described here:

- From ***integer*** to ***number***: an *integer* is provided when a *number* is expected (for example, an *integer* and a *number* are passed as inputs of a n-ary numeric operator); it returns a *number* having the integer part equal to the *integer* and the decimal part equal to zero;
- From ***integer*** to ***string***: an *integer* is provided when a *string* is expected (for example, an *integer* is passed as an input of a *string* operator); it returns a *string* having the literal value of the *integer*;
- From ***number*** to ***string***: a *number* is provided when a *string* is expected; it returns the *string* having the literal value of the *number*; the decimal separator is converted into the character "." (dot).
- From ***boolean*** to ***string***: a *boolean* is provided when a *string* is expected; the boolean value TRUE is converted into the *string* "TRUE" and FALSE into the *string* "FALSE";
- From ***date*** to ***time***: a *date* (point in time) is provided when a *time* is expected (interval of time): the conversion results in an interval having the same start and end, both equal to the original *date*;
- From ***time_period*** to ***time***: a *time_period* (a regular interval of *time*, like a month, a quarter, a year …) is provided when a *time* (any interval of time) is expected; it returns a *time* value having the same start and end as the *time_period* value.

An implicit cast is also performed from a **value domain type** or a **set type** to a **basic scalar type**: when a *scalar* value belonging to a Value Domains or a Set is involved in an operation (i.e., provided as input to an operator), the value is implicitly cast into the basic scalar type which the Value Domain refers to (for this relationship, see the description of Type System in the User Manual). For example, assuming that the Component birth_country is defined on the Value Domain country, which contains the ISO 3166-1 numeric codes and therefore refers to the basic scalar type *integer*, the (possible) invocation length(birth_country), which calculates the length of the input string, automatically casts the values of birth_country into the corresponding string. If the basic scalar type of the Value Domain is not compatible with the expression where it is used, an error is raised. This VTL feature is particularly important as it provides a general behaviour for the Value Domains and relevant Sets, preventing from the need of defining specific behaviours (or methods or operations) for each one of them. In other words, all the Values inherit the operations that can be performed on them from the basic scalar types of the respective Value Domains.

The **cast** operator can be invoked explicitly even for the conversions which allow an implicit cast and in this case the same behaviour as the implicit cast is applied.

The behaviour of the **cast** operator for the conversions that require **explicit casting without mask** is the following:

- From ***integer*** to ***boolean***: if the *integer* is different from 0, then TRUE is returned, FALSE otherwise.
- From ***number*** to ***integer***: converts a *number* with no decimal part into an *integer*; if the decimal part is present, a runtime error is raised.
- From ***number*** to ***boolean***: if the *number* is different from 0.0, then TRUE is returned, FALSE otherwise.

- From ***boolean*** to ***integer***: TRUE is converted into 1; FALSE into 0.
- From ***boolean*** to ***number***: TRUE is converted into 1.0; FALSE into 0.0.
- From ***date*** to ***time_period*:** it converts a *date* into the corresponding daily value of *time_period*.
- From ***string*** to ***integer*:** the *integer* having the literal value of the *string* is returned; if the *string* contains a literal that cannot be matched to an *integer*, a runtime error is raised.
- From ***string*** to ***time_period***: it converts a *string* value to a *time_period* value.

When an **explicit casting with mask** is required, the conversion is made by applying the formatting mask which specifies the meaning of the characters in the output *string*. The formatting Masks are described in the section "VTL-ML – Typical Behaviour of the ML Operators", sub-section "Type Conversion and Formatting Mask.

The behaviour of the **cast** operator for such conversions is the following:

- From ***time*** to ***string***: it is applied the *time* formatting mask.
- From ***date*** to ***string*:** it is applied the *time_period* formatting mask.
- From ***time_period*** to ***date***: it is applied a formatting mask which accepts two possible values ("START", "END"). If "START" is specified, then the *date* is set to the beginning of the *time_period*; if "END" is specified, then the *date* is set to the end of the *time_period*.
- From ***time_period*** to ***string***: it is applied the *time_period* formatting mask.
- *From **duration** to **string**:* a *duration* (an absolute time interval) is provided when a *string* is expected; it returns the *string* having the default *string* representation for the *duration.*
- From ***string*** to ***number*:** the *number* having the literal value of the *string* is returned; if the *string* contains a literal that cannot be matched to a *number*, a runtime error is raised. The *number* is generated by using a *number* formatting mask.
- From ***string*** to ***time*:** the *time* having the literal value of the *string* is returned; if the *string* contains a literal that cannot be matched to a *date*, a runtime error is raised. The *time* value is generated by using a *time* formatting mask.
- From ***string*** to ***duration*:** the *duration* having the literal value of the *string* is returned; if the *string* contains a literal that cannot be matched to a *duration*, a runtime error is raised. The *duration* value is generated by using a time formatting mask.

Conversions between basic scalar types and Value Domains or Set types

A value of a basic *scalar* type can be converted into a value belonging to a Value Domain which refers to such a *scalar* type. The resulting *scalar* value must be one of the allowed values of the Value Domain or Set; otherwise, a runtime error is raised. This specific use of **cast** operators does not really correspond to a type conversion; in more formal terms, we would say that it acts as a constructor, i.e., it builds an instance of the output type. Yet, towards a homogeneous and possibly simple definition of VTL syntax, we blur the distinction between constructors and type conversions and opt for a unique formalism. An example is given below.

Conversions between different Value Domain types

As a result of the above definitions, conversions between values of different Value Domains are also possible. Since an element of a Value Domain is implicitly cast into its corresponding basic scalar type, we can build on it to turn the so obtained scalar type into another Value Domain type. Of course, this latter Value Domain type must use as a base type this scalar type.

Examples

Example 1: from *string* to *number*

```
ds2 := ds1[calc m2 := cast(m1, number, "DD.DDD") + 2) ]
```

In this case we use explicit cast from *string* to *numbers*. The mask is used to specify how the *string* must be interpreted in the conversion.

Example 2: from *string* to *date*

```
ds2 := ds1[calc m2 := cast(m1, date, "YYYY-MM-DD") ]
```

In this case we use explicit cast from *string* to *date*. The mask is used to specify how the *string* must be interpreted in the conversion.

Example 3: from *number* to *integer*

```
ds2 := ds1[calc m2 := cast(m1, integer) + 3 ]
```

In this case we cast a *number* into an *integer*, no mask is required.

Example 4: from *number* to *string*

```
ds2 := ds1[calc m2 := length(cast(m1, string)) ]
```

In this case we cast a *number* into a *string*, no mask is required.

Example 5: from *date* to *string*

```
ds2 := ds1[calc m2 := cast(m1, string, “YY-MON-DAY hh:mm:ss”) ]
```

In this example a *date* instant is turned into a *string*. The mask is used to specify the *string* layout.

Example 6: from *string* to *GEO_AREA*

```
ds2 := ds1[calc m2 := cast(GEO_STRING, GEO_AREA)]
```

In this example we suppose we have elements of Value Domain Subset for GEO_AREA. Let GEO_STRING be a string Component of Data Set ds1 with string values compatible with the GEO_AREA Value Domain Subset. Thus, the following expression moves ds1 data into ds2, explicitly casting strings to geographical areas.

Example 7: from *GEO_AREA* to *string*

```
ds2 := ds1[calc m2 := length(GEO_AREA)]
```

In this example we use a Component GEO_AREA in a *string* expression, which calculates the length of the corresponding *string*; this triggers the automatic cast.

Example 8: from *GEO_AREA2* to *GEO_AREA1*

```
ds2 := ds1 [ calc m2 := cast (GEO, GEO_AREA1) ]
```

In this example we suppose we have to compare elements two Value Domain Subsets, They are both defined on top of Strings. The following cast expressions performs the conversion.

Now, Component GEO is of type GEO_AREA2, then we specify it has to be cast into GEO_AREA1. As both work on *strings* (and the values are compatible), the conversion is feasible. In other words, the cast of an operand into GEO_AREA1 would expect a *string*. Then, as GEO is of type GEO_AREA2, defined on top of *strings*, it is implicitly cast to the respective *string*; this is compatible with what cast expects and it is then able to build a value of type GEO_AREA1.

Example 9: from *string* to *time_period*

In the following examples we convert from *strings* to *time_periods*, by using appropriate masks.

The first quarter of year 2000 can be expressed as follows (other examples are possible):

```
cast ( “2000Q1”, time_period, “YYYY\QQ” )
cast ( “2000-Q1”, time_period, “YYYY-\QQ” )
cast ( “2000-1”, time_period, “YYYY-Q” )
cast ( “Q1-2000”, time_period, “\QQ-YYYY” )
cast ( “2000Q01”, time_period, “YYYY\QQQ” )
```

Examples of daily data:

```
cast ( “2000M01D01”, time_period, “YYYY\MMM\DDD” )
cast ( “2000.01.01”, time_period, “YYYY\.MM\.DD” )
```

VTL-ML - Join operators

The Join operators are fundamental VTL operators. They are part of the core of the language and allow to obtain the behaviour of the majority of the other non-core operators, plus many additional behaviours that cannot be obtained through the other operators.
The Join operators are four, namely the inner_join, the left_join, the full_join and the cross_join. Because their syntax is similar, they are described together.

Join : inner_join, left_join, full_join, cross_join

Syntax

joinOperator **(** ds1 { **as** alias1 } { **,** dsN { **as** aliasN } }* { **using** usingComp { **,** usingComp }* }
{ **filter** filterCondition }
{ **apply** applyExpr
| **calc** calcClause
| **aggr** aggrClause { groupingClause } }
{ **keep** comp {, comp }* | **drop** comp {, comp }* }
{ **rename** compFrom **to** compTo { **,** compFrom **to** compTo }* }
)

joinOperator ::= { **inner_join** | **left_join** | **full_join** | **cross_join** }^{1}

calcClause ::= { calcRole } calcComp **:=** calcExpr
{ , { calcRole } calcComp **:=** calcExpr }*

calcRole ::= {**identifier** | **measure** | **attribute** | **viral attribute**}^{1}

aggrClause ::= { aggrRole } aggrComp **:=** aggrExpr
{ **,** { aggrRole } aggrComp **:=** aggrExpr }*

aggrRole ::= { **measure** | **attribute** | **viral attribute** }^{1}

groupingClause ::= { **group by** groupingId { **,** groupingId }*
| **group except** groupingId { **,** groupingId }*
| **group all** conversionExpr }^{1}
{ **having** havingCondition }

Input parameters

joinOperator	the Join operator to be applied
ds1, …, dsN	the Data Set operands (at least one must be present)
alias1, …, aliasN	optional aliases for the input Data Sets, valid only within the “join” operation to make it easier to refer to them. If omitted, the Data Set name must be used.
usingComp	component of the input Data Sets whose values have to match in the join (the **using** clause is allowed for the **left_join** only under certain constraints described below and is not allowed at all for the **full_join** and **cross_join**)
filterCondition	a condition (*boolean* expression) at component level, having only Components of the input Data Sets as operands, which is evaluated for each joined Data Point and filters them (when TRUE the joined Data Point is kept, otherwise it is not kept)
applyExpr	an expression, having the input Data Sets as operands, which is pairwise applied to all their homonym Measure Components and produces homonym Measure Components in the result; for example if both the Data Sets ds1 and ds2 have the *numeric* measures m1 and m2, the clause apply ds1 + ds2 would result in calculating m1 := ds1#m1 + ds2#m1 and m2 := ds1#m2 + ds2#m2
calcClause	clause that specifies the Components to be calculated, their roles and their calculation algorithms, to be applied on the joined and filtered Data Points.
calcRole	the role of the Component to be calculated
calcComp	the name of the Component to be calculated

calcExpr — expression at component level, having only Components of the input Data Sets as operands, used to calculate a Component

aggrClause — clause that specifies the required aggregations, i.e., the aggregated Components to be calculated, their roles and their calculation algorithm, to be applied on the joined and filtered Data Points

aggrRole — the role of the aggregated Component to be calculated; if omitted, the Measure role is assumed

aggrComp — the name of the aggregated Component to be calculated; this is a dependent Component of the result (Measure or Attribute, not Identifier)

aggrExpr — expression at component level, having only Components of the input Data Sets as operands, which invokes an aggregate operator (e.g. **avg**, **count**, **max** … , see also the corresponding sections) to perform the desired aggregation. Note that the **count** operator is used in an aggrClause without parameters, e.g.:

```
DS_1 [ aggr Me_1 := count ( ) group by Id_1 ) ]
```

groupingClause — the following alternative grouping options:

- **group by** — the Data Points are grouped by the values of the specified Identifiers (groupingId). The Identifiers not specified are dropped in the result.
- **group except** — the Data Points are grouped by the values of the Identifiers not specified as groupingId. The specified Identifiers are dropped in the result.
- **group all** — converts the values of an Identifier Component using conversionExpr and keeps all the resulting Identifiers.

groupingId — Identifier Component to be kept (in the **group by** clause) or dropped (in the **group except** clause).

conversionExpr — specifies a conversion operator (e.g. **time_agg**) to convert an Identifier from finer to coarser granularity. The conversion operator is applied on an Identifier of the operand Data Set op.

havingCondition — a condition (*boolean* expression) at component level, having only Components of the input Data Sets as operands (and possibly constants), to be fulfilled by the groups of Data Points: only groups for which havingCondition evaluates to TRUE appear in the result. The havingCondition refers to the groups specified through the groupingClause, therefore it must invoke aggregate operators (e.g. avg, count, max, …, see also the section Aggregate invocation). A correct example of havingCondition is max(obs_value) < 1000, while the condition obs_value < 1000 is not a right havingCondition, because it refers to the values of single Data Points and not to the groups. The count operator is used in a havingCondition without parameters, e.g.:

```
sum ( ds group by id1 having count ( ) >= 10 )
```

comp — dependent Component (Measure or Attribute, not Identifier) to be kept (in the **keep** clause) or dropped (in the **drop** clause)

compFrom — the original name of the Component to be renamed

compTo — the new name of the Component atfer the renaming

Examples of valid syntaxes

```
inner_join  ( ds1 as d1,  ds2 as d2 using Id1, Id2
                    filter d1#Me1 + d2#Me1 <10
                    apply  d1 / d2
                    keep Me1, Me2, Me3
                    rename Id1 to Id10, id2 to id20
                   )

left_join ( ds1 as d1,  ds2 as d2
               filter d1#Me1 + d2#Me1 <10,
               calc Me1 := d1#Me1 + d2#Me3,
               keep  Me1
               rename Id1 to Ident1, Me1 to Meas1
              )

full_join ( ds1 as d1,  ds2 as d2
               filter d1#Me1 + d2#Me1 <10,
```

```
            aggr Me1 := sum(Me1), attribute At20 := avg(Me2)
            group by  Id1, Id2
        having sum(Me3) > 0
        )
```

Semantics for scalar operations

The join operator does not perform scalar operations.

Input parameters type

ds1, …, dsN ::	dataset
alias1, …, aliasN ::	name
usingId ::	name < component >
filterCondition ::	component<boolean>
applyExpr ::	dataset
calcComp ::	name < component >
calcExpr ::	component<scalar>
aggrComp ::	name < component >
aggrExpr ::	component<scalar>
groupingId ::	name < identifier >
conversionExpr ::	component<scalar>
havingCondition ::	component<boolean>
comp ::	name < component >
compFrom ::	component<scalar>
compTo ::	component<scalar>

Result type

result ::	dataset

Additional constraints

The aliases must be all distinct and different from the Data Set names. Aliases are mandatory for Data Sets which appear more than once in the Join (self-join) and for non-named Data Set obtained as result of a sub-expression.

The using clause is not allowed for the **full_join** and for the **cross_join**, because otherwise a non-functional result could be obtained.

If the using clause is not specified (we will label this case as "Case A"), calling $Id(ds_i)$ the set of Identifier Components of operand ds_i, the following group of constraints must hold[7]:

- For **inner_join**, for each pair ds_i, ds_j, either $Id(ds_i) \subseteq Id(ds_j)$ or $Id(ds_j) \subseteq Id(ds_i)$. In simpler words, the Identifiers of one of the joined Data Sets must be a superset of the identifiers of all the other ones.
- For **left_join** and **full_join**, for each pair ds_i, ds_j, $Id(ds_i) = Id(ds_j)$. In simpler words, the joined Data Sets must have the same Identifiers.
- For **cross-join** (Cartesian product), no constraints are needed.

If the using clause is specified (we will label this case as "Case B", allowed only for the **inner_join** and the **left_join**), all the join keys must appear as Components in all the input Data Sets. Moreover two sub-cases are allowed:

- Sub-case B1: the constraints of the Case A are respected and the join keys are a subset of the common Identifiers of the joined Data Sets;
- Sub-case B2:
  - In case of **inner_join**, one Data Set acts as the reference Data Set which the others are joined to; in case of **left_join**, this is the "more to the left" Data Set (i.e., ds_1);
  - All the input Data Sets, except the reference Data Set, have the same Identifiers $[Id_1, \ldots, Id_n]$;
  - The using clause specifies all and only the common Identifiers of the non-reference Data Sets $[Id_1, \ldots, Id_n]$.

The join operators must fulfil also other constraints:

- **apply**, **calc** and **aggr** clauses are mutually exclusive
- **keep** and **drop** clauses are mutually exclusive
- comp can be only dependent Components (Measures and Attributes, not Identifiers)
- An Identifier not included in the **group by** clause (if any) cannot be included in the **rename** clause

[7] These constraints hold also for the **full_join** and the **cross_join**, which do not allow the using clause.

- An Identifier included in the **group except** clause (if any) cannot be included in the **rename** clause. If the **aggr** clause is invoked and the grouping clause is omitted, no Identifier can be included in the **rename** clause
- A dependent Component not included in the **keep** clause (if any) cannot be renamed
- A dependent Component included in the **drop** clause (if any) cannot be renamed

Behaviour

The **semantics of the join operators** can be procedurally described as follows.

- A relational join of the input operands is performed, according to SQL inner (**inner_join**), left-outer (**left_join**), full-outer (**full_join**) and Cartesian product (**cross_join**) semantics (these semantics will be explained below), producing an intermediate internal result, that is a Data Set that we will call "virtual" (VDS_1).
- The filterCondition, if present, is applied on VDS_1, producing the Virtual Data Set VDS_2.
- The specified calculation algorithms (**apply**, **calc** or **aggr**), if present, are applied on VDS_2. For the Attributes that have not been explicitly calculated in these clauses, the Attribute propagation rule is applied (see the User Manual), so producing the Virtual Data Set VDS_3.
- The **keep** or **drop** clause, if present, is applied on VDS_3, producing the Virtual Data Set VDS_4.
- The **rename** clause, if present, is applied on VDS_4, producing the Virtual Data Set VDS_5.
- The final automatic alias removal is performed in order to obtain the output Data Set.

An alias can be optionally declared for each input Data Set. The aliases are valid only within the "join" operation, in particular to allow joining a dataset with itself (self join). If omitted, the input Data Sets are referenced only through their Data Set names. If the aliases are ambiguous (for example duplicated or equal to the name of another Data Set), an error is raised.

The **structure of the virtual Data Set** VDS_1 which is the output of the relational join is the following.

For the **inner_join**, the **left_join** and the **full_join**, the virtual Data Set contains the following Components:

- The Components used as join keys, which appear once and maintain their original names and roles. In the cases A and B1, all of them are Identifiers. In the sub-case B2, the result takes the roles from the reference Data Set.
- In the sub-case B2: the Identifiers of the reference Data Set, which appear once and maintain their original name and role.
- The other Components coming from exactly one input Data Set, which appear once and maintain their original name
- The other Components coming from more than one input Data Set, which appears as many times as the Data Set they come from; to distinguish them, their names are prefixed with the alias (or the name) of the Data Set they come from, separated by the "#" symbol (e.g., ds_i#cmp_j). For example, if the Component "population" appears in two input Data Sets "ds1" and "ds2" that have the aliases "a" and "b" respectively, the Components "a#population" and "b#population" will appear in the virtual Data Set. If the aliases are not defined, the two Components are prefixed with the Data Set name (i.e., "ds1#population" and "ds2#population"). In this context, the symbol "#" does not denote the membership operator but acts just as a separator between the the Data Set and the Component names.
- If the same Data Set appears more times as operand of the join (self-join) and the aliases are not defined, an exception is raised because it is not allowed that two or more Components in the virtual Data Set have the same name. In the self-join the aliases are mandatory to disambiguate the Component names.
- If a Data Set in the join list is the result of a sub-expression, then an alias is mandatory all the same because this Data Set has no name. If the alias is omitted, an exception is raised.

As for the **cross_join**, the virtual Data Set contains all the Components from all the operands, possibly prefixed with the aliases to avoid ambiguities.

The **semantics of the relational join** is the following.

The join is performed on some join keys, which are the Components of the input Data Sets whose values are used to match the input Data Points and produce the joined output Data Points.

By default (only for the **full_join** and the **cross_join**), the join is performed on the subset of homonym Identifier Components of the input Data Sets.

The parameter **using** allows to specify different join keys than the default ones, and can be used only for the **inner_join** and the **left_join** in order to preserve the functional behaviour of the operations.

The different kinds of relational joins behave as follows.

- **inner_join**: the Data Points of ds1, …, dsN are joined if they have the same values for the common Identifier Components or, if the **using** clause is present, for the specified Components. A (joined) virtual Data Point is generated in the virtual Data Set VDS_1 when a matching Data Point is found for each one of the input Data Sets. In this case, the Values of the Components of a virtual Data Point are taken from the corresponding Components of the matching Data Points. If there is no match for one or more input Data Sets, no virtual Data Point is generated.

- **left_join**: the join is ideally performed stepwise, between consecutive pairs of input Data Sets, starting from the left side and proceeding towards the right side. The Data Points are matched like in the **inner_join**, but a virtual Data Point is generated even if no Data Point of the right Data Set matches (in this case, the Measures and Attributes coming from the right Data Set take the NULL value in the virtual Data Set). Therefore, for each Data Points of the left Data Set a virtual Data Point is always generated. These stepwise operations are associative. More formally, consider the generic pair $<ds_i, ds_{i+1}>$, where ds_i is the result of the left_join of the first "i" operands and ds_{i+1} is the i+1[th] operand. For each pair $<ds_i, ds_{i+1}>$, the joined Data Set is fed with all the Data Points that match in ds_i and ds_{i+1} or are only in ds_i. The constraints described above guarantee the absence of null values for the Identifier Components of the joined Data Set, whose values are always taken from the left Data Set. If the join succeeds for a Data Point in ds_i, the values for the Measures and the Attributes are carried from ds_i and ds_{i+1} as explained above. Otherwise, i.e., if no Data Point in ds_{i+1} matches the Data Point in ds_i, null values are given to Measures and Attributes coming only from ds_{i+1}.
- **full_join**: the join is ideally performed stepwise, between consecutive pairs of input Data Sets, starting from the left side and proceeding toward the right side. The Data Points are matched like in the **inner_join** and **left_join**, but the **using** clause is not allowed and a virtual Data Point is generated either if no Data Point of the right Data Set matches with the left Data Point or if no Data Point of the left Data Set matches with the right Data Point (in this case, Measures and Attributes coming from the non matching Data Set take the NULL value in the virtual Data Set). Therefore, for each Data Points of the left and the right Data Set, a virtual Data Point is always generated. These stepwise operations are associative. More formally, consider the generic pair $<ds_i, ds_{i+1}>$, where ds_i is the result of the **full_join** of the first "i" operands and ds_{i+1} is the i+1[th] operand. For each pair $<ds_i, ds_{i+1}>$, the resulting Data Set is fed with the Data Points that match in ds_i and ds_{i+1} or that are only in ds_i or in ds_{i+1}. If for a Data Point in ds_i the join succeeds, the values for the Measures and the Attributes are carried from ds_i and ds_{i+1} as explained. Otherwise, i.e., if no Data Point in ds_{i+1} matches the Data Point in ds_i, NULL values are given to Measures and Attributes coming only from ds_{i+1}. Symmetrically, if no Data Point in ds_i matches the Data Point in ds_{i+1}, NULL values are given to Measures and Attributes coming only from ds_i. The constraints described above guarantee the absence of NULL values on the Identifier Components. As mentioned, the **using** clause is not allowed in this case.
- **cross_join**: the join is performed stepwise, between consecutive pairs of input Data Sets, starting from the left side and proceeding toward the right side. No match is performed but the Cartesian product of the input Data Points is generated in output. These stepwise operations are associative. More formally, consider the ordered pair $<ds_i, ds_{i+1}>$, where ds_i is the result of the cross_ join of the first "i" operands and ds_{i+1} is the i+1-th operand. For each pair $<ds_i, ds_{i+1}>$, the resulting Data Set is fed with the Data Points obtained as the Cartesian product between the Data Points of ds_i and ds_{i+1}. The resulting Data Set will have all the Components from ds_i and ds_{i+1}. For the Data Sets which have at least one Component in common, the alias parameter is mandatory. As mentioned, the **using** parameter is not allowed in this case.

The **semantics of the clauses** is the following.

- **filter** takes as input a Boolean Component expression (having type *component<boolean>*). This clause filters in or out the input Data Points; when the expression is TRUE the Data Point is kept, otherwise it is not kept in the result. Only one **filter** clause is allowed.
- **apply** combines the homonym Measures in the source operands whose type is compatible with the operators used in applyExpr, generating homonym Measures in the ouput. The expression applyExpr can use as input the names or aliases of the operand Data Sets. It applies the expression to all the n-uples of homonym Measures in the input Data Sets producing in the target a single homonym Measure for each n-uple. It can be thought of as the multi-measure version of the **calc**. For example, if the following aliases have been declared: d1, d2, d3, then the following expression d1+d2+d3, sums all the homonym Measures in the three input Data Sets, say M1 and M2, so as to obtain in the result: M1 := d1#M1 + d2#M1 + d3#M1 and M2 := d1#M2 + d2#M2 + d3#M2. It is not only a compact version of a multiple **calc**, but also essential when the number of Measures in the input operands is not known beforehand. Only one **apply** clause is allowed.
- **calc** calculates new Identifier, Measure or Attribute Components on the basis of sub-expressions at Component level. Each Component is calculated through an independent sub-expression. It is possible to specify the role of the calculated Component among **measure**, **identifier**, **attribute**, or **viral attribute**, therefore the **calc** clause can be used also to change the role of a Component when possible. The keyword **viral** allows controlling the virality of Attributes (for the Attribute propagation rule see the User Manual). The following rule is used when the role is omitted: if the component exists in the operand Data Set then it maintains that role; if the component does not exist in the operand Data Set then the role is **measure**. The calcExpr are independent one another, they can only reference

Components of the input Virtual Data Set and cannot use Components generated, for example, by other calcExpr . If the calculated Component is a new Component, it is added to the output virtual Data Set. If the Calculated component is a Measure or an Attribute that already exists in the input virtual Data Set, the calculated values overwrite the original values. If the Calculated component is an Identifier that already exists in the input virtual Data Set, an exception is raised because overwriting an Identifier Component is forbidden for preserving the functional behaviour. Analytic operators can be used in the **calc** clause.

- **aggr** calculates aggregations of dependent Components (Measures or Attributes) on the basis of sub-expressions at Component level. Each Component is calculated through an independent sub-expression. It is possible to specify the role of the calculated Component among **measure**, **identifier**, **attribute**, or **viral attribute**. The substring **viral** allows to control the virality of Attributes, if the Attribute propagation rule is adopted (see the User Manual). The **aggr** sub-expressions are independent of one another, they can only reference Components of the input Virtual Data Set and cannot use Components generated, for example, by other **aggr** sub-expressions. The **aggr** computed Measures and Attributes are the only Measures and Attributes returned in the output virtual Data Set (plus the possible viral Attributes, see below **Attribute propagation**). The sub-expressions must contain only Aggregate operators, which are able to compute an aggregated Value relevant to a group of Data Points. The groups of Data Points to be aggregated are specified through the groupingClause, which allows the following alternative options.

group by	the Data Points are grouped by the values of the specified Identifier. The Identifiers not specified are dropped in the result.
group except	the Data Points are grouped by the values of the Identifiers not specified in the clause. The specified Identifiers are dropped in the result.
group all	converts an Identifier Component using conversionExpr and keeps all the resulting Identifiers.

  The **having** clause is used to filter groups in the result by means of an aggregate condition evaluated on the single groups, for example the minimum number of rows in the group.

  If no grouping clause is specified, then all the input Data Points are aggregated in a single group and the clause returns a Data Set that contains a single Data Point and has no Identifier Components.
- **keep** maintains in the output only the specified dependent Components (Measures and Attributes) of the input virtual Data Set and drops the non-specified ones. It has the role of a projection in the usual relational semantics (specifying which columns have to be projected in). Only one **keep** clause is allowed. If **keep** is used, **drop** must be omitted.
- **drop** maintains in the output only the non-specified dependent Components (Measures and Attributes) of the input virtual Data Set (component<scalar>) and drops the specified ones. It has the role of a projection in the usual relational join semantics (specifying which columns will be projected out). Only one **drop** clause is allowed. If **drop** is used, **keep** must be omitted.
- **rename** assigns new names to one or more Components (Identifier, Measure or Attribute Components). The resulting Data Set, after renaming all the specified Components, must have unique names of all its Components (otherwise a runtime error is raised). Only the Component name is changed and not the Component Values, therefore the new Component must be defined on the same Value Domain and Value Domain Subset as the original Component (see also the IM in the User Manual). If the name of a Component defined on a different Value Domain or Set is assigned, an error is raised. In other words, rename is a transformation of the variable without any change in its values.

The semantics of the **Attribute propagation** in the join is the following. The Attributes calculated through the **calc** or **aggr** clauses are maintained unchanged. For all the other Attributes that are defined as **viral**, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule section in the User Manual). This is done before the application of the **drop**, **keep** and **rename** clauses, which acts also on the Attributes resulting from the propagation.

The semantics of the **final automatic aliases** removal is the following. After the application of all the clauses, the structure of the final virtual Data Set is further modified. All the Components of the form "alias#component_name" (or "dataset_name#component_name") are implicitly renamed into "component_name". This means that the prefixes in the Component names are automatically removed. It is responsibility of the user to guarantee the absence of duplicated Component names once the prefixes are removed. In other words, the user must ensure that there are no pairs of Components whose names are of the form "alias1#c1" and "alias2#c1" in the structure of the virtual Data Point, since the removal of "alias1" and "alias2" would cause the clash. If, after the aliases removal two Components have the same name, an error is raised. In particular, name conflicts may derive if the using clause is present and some homonym Identifier Components do not appear in it; these components should be properly renamed because cannot be removed; the input Data Set have homonym Measures and there is no apply clause which unifies them; these Measures can be renamed or removed.

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	A	B
1	B	C	D
2	A	E	F

DS_2			
Id_1	**Id_2**	**Me_1A**	**Me_2**
1	A	B	Q
1	B	S	T
3	A	Z	M

Example 1:

```
DS_r := inner_join ( DS_1 as d1, DS_2 as d2,
        keep Me_1, d2#Me_2, Me_1A)
```
results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_1A**
1	A	A	Q	B
1	B	C	T	S

Example 2:

```
DS_r := left_join ( DS_1 as d1, DS_2 as d2,
        keep Me_1, d2#Me_2, Me_1A )
```
results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_1A**
1	A	A	Q	B
1	B	C	T	S
2	A	E	null	null

Example 3:

```
DS_r := full_join ( DS_1 as d1, DS_2 as d2,
        keep Me_1, d2#Me_2, Me_1A )
```
results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_1A**
1	A	A	Q	B
1	B	C	T	S
2	A	E	null	null

3	A	null	M	Z

Example 4:

```
DS_r := cross_join (DS_1 as d1, DS_2 as d2,
        rename d1#Id_1 to Id11, d1#Id_2 to Id12, d2#Id1 to Id21, d2#Id2 to Id22, d1#Me_2
        to Me12 )
```

results in:

DS_r							
Id_11	**Id_12**	**Id_21**	**Id_22**	**Me_1**	**Me12**	**Me_1A**	**Me_2**
1	A	1	A	A	B	B	Q
1	A	1	B	A	B	S	T
1	A	3	A	A	B	Z	M
1	B	1	A	C	D	B	Q
1	B	1	B	C	D	S	T
1	B	3	A	C	D	Z	M
2	A	1	A	E	F	B	Q
2	A	1	B	E	F	S	T
2	A	3	A	E	F	Z	M

Example 5:

```
DS_r := inner_join (DS_1 as d1, DS_2 as d2,
        filter Me_1 = "A",
        calc Me_4 = Me_1 || Me_1A,
        drop d1#Me_2)
```

where || is the string concatenation, results in**:**

DS_r					
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_1A**	**Me_4**
1	A	A	Q	B	AB

Example 6:

```
DS_r := inner_join ( DS_1
        calc Me_2 := Me_2 || "_NEW"
        filter Id_2 ="B"
        keep Me_1, Me_2)
```

where || is the string concatenation, results in**:**

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	B	C	D_NEW

Example 7:

Given the operand Data Sets DS_1 and DS_2:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	A	B
1	B	C	D
2	A	E	F

DS_2			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	B	Q
1	B	S	T
3	A	Z	M

```
DS_r  := inner_join ( DS_1 as d1, DS_2 as d2,
        apply d1 || d2)
```

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	AB	BQ
1	B	CS	DT

VTL-ML - String operators

String concatenation : | |

Syntax

op1 || op2

Input Parameters

op1, op2 the operands

Examples of valid syntaxes

"Hello" || ", world!"
ds_1 || ds_2

Semantics for scalar operations

Concatenates two strings. For example, "Hello" || ", world!" gives "Hello, world!"

Input parameters type

op1, op2 :: dataset { measure<string> _+ }
| component<string>
| string

Result type

result :: dataset { measure<string> _+ }
| component<string>
| string

Additional constraints

None.

Behaviour

The operator has the behaviour of the "Operators applicable on two Scalar Values or Data Sets or Data Set Components" (see the section "Typical behaviours of the ML Operators").

Examples

Given the Data_Sets DS_1 and DS_2:

DS_1		
Id_1	**Id_2**	**Me_1**
1	A	"hello"
2	B	"hi"

DS_2		
Id_1	**Id_2**	**Me_1**
1	A	"world"
2	B	"there"

Example 1: DS_r := DS_1 || DS_2 results in:

DS_r		
Id_1	**Id_2**	**Me_1**
1	A	"helloworld"
2	B	"hithere"

Example 2 (on component): DS_r := DS_1[calc Me_2:= Me_1 **||** " world"] results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	"hello"	"hello world"
2	B	"hi"	"hi world"

Whitespace removal : trim, rtrim, ltrim

Syntax

{**trim|ltrim|rtrim**}^{1} **(** op **)**

Input parameters

op the operand

Examples of valid syntaxes

trim("Hello ")
trim(ds_1)

Semantics for scalar operations

Removes trailing or/and leading whitespace from a string. For example, trim("Hello ") gives "Hello".

Input parameters type

op :: dataset { measure<string> _+ }
| component<string>
| string

Result type

result :: dataset { measure<string> _+ }
| component<string>
| string

Additional constraints

None.

Behaviour

The operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component" (see the section "Typical behaviours of the ML Operators").

Examples

Given the Data Set DS_1:

DS_1		
Id_1	**Id_2**	**Me_1**
1	A	"hello "
2	B	"hi "

Example 1: DS_r := rtrim(DS_1) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
1	A	"hello"
2	B	"hi"

Example 2 (on component): DS_r := DS_1[calc Me_2:= rtrim(Me_1)] results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	"hello "	"hello"
2	B	"hi "	"hi"

Character case conversion : upper/lower

Syntax

{**upper** | **lower**}[1] **(** op **)**

Input Parameters

op the operand

Examples of valid syntaxes

upper("Hello")
lower(ds_1)

Semantics for scalar operations

Converts the character case of a string in upper or lower case. For example, upper("Hello") gives "HELLO".

Input Parameters type

op :: dataset { measure<string> _+ }
| component<string>
| string

Result type

result :: dataset { measure<string> _+ }
| component<string>
| string

Additional constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the Data Set DS_1:

DS_1		
Id_1	**Id_2**	**Me_1**
1	A	"hello"
2	B	"hi"

Example 1: DS_r := upper(DS_1)   results in:

DS_r		
Id_1	**Id_2**	**Me_1**
1	A	"HELLO"
2	B	"HI"

Example 2 (on component): DS_r := DS_1[calc Me_2:= upper(Me_1)]   results in:

DS_R			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	"hello"	"HELLO"
2	B	"hi"	"HI"

Sub-string extraction :   **substr**

Syntax

substr (op, *start*, *length* **)**

Input parameters

op   the operand
start   the starting digit (first character) of the string to be extracted
length   the length (number of characters) of the string to be extracted

Examples of valid syntaxes

substr (DS_1, 2 , 3)
substr (DS_1, 2)
substr (DS_1, _ , 3)
substr (DS_1)

Semantics for scalar operations

The operator extracts a substring from op, which must be *string* type. The substring starts from the startth character of the input string and has a number of characters equal to the length parameter.

- If start is omitted, the substring starts from the 1st position.
- If length is omitted or overcomes the length of the input string, the substring ends at the end of the input string.
- If start is greater than the length of the input string, an empty string is extracted.

For example:
substr (“abcdefghijklmnopqrstuvwxyz”, start:= 5 , length:= 10)   gives: “efghijklmn”.
substr (“abcdefghijklmnopqrstuvwxyz”, start:= 25 , length:= 10)   gives: “yz”.
substr (“abcdefghijklmnopqrstuvwxyz”, start:= 30 , length:= 10)   gives: “”.

Input parameters type

op ::   dataset { measure <string> _+ }
  | component <string>
  | string

start ::   component < integer [value >= 1] >
  | integer [value >= 1]

length :: component < integer [value >= 0] >
| integer [value >= 0]

Result type

result :: dataset { measure<string> _+ }
| component<string>
| string

Additional constraints

None.

Behaviour

As for the invocations at Data Set level, the operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component", as for the invocations at Component or Scalar level, the operator has the behaviour of the "Operators applicable on more than two Scalar Values or Data Set Components", (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	"hello world"	"medium size text"
1	B	"abcdefghilmno"	"short text"
2	A	"pqrstuvwxyz"	"this is a long description"

Example 1: DS_r:= substr (DS_1 , 7) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	"world"	" size text"
1	B	"ghilmno"	"text"
2	A	"vwxyz"	"s a long description"

Example 2: DS_r:= substr (DS_1 , 1 , 5) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	"hello"	"mediu"
1	B	"abcde"	"short"
2	A	"pqrst"	"this "

Example3(on Components): DS_r:= DS_1 [calc Me_2:= substr (Me_2 , 1 , 5)] results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	"hello world"	"mediu"

1	B	"abcdefghilmno"	"short"
2	A	"pqrstuvwxyz"	"this "

String pattern replacement: **replace**

Syntax

replace (op , pattern1, *pattern2* **)**

Input parameters

op the operand
pattern1 the pattern to be replaced
pattern2 the replacing pattern

Examples of valid syntaxes

replace(DS_1, "Hello", "Hi")
replace(DS_1, "Hello")

Semantics for scalar operations

Replaces all the occurrences of a specified string-pattern (pattern1) with another one (pattern2). If pattern2 is omitted then all occurrences of pattern1 are removed. For example:

replace("Hello world", "Hello", "Hi") gives "Hi world"
replace("Hello world", "Hello") gives " world"
replace ("Hello", "ello", "i") gives "Hi"

Input parameters type

op :: dataset { measure<string> _+ }
| component<string>
| string
pattern1, pattern2 :: component<string>
| string

Result type

result :: dataset { measure<string> _+ }
| component<string>
| string

Additional constraints

None.

Behaviour

As for the invocations at Data Set level, the operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component", as for the invocations at Component or Scalar level, the operator has the behaviour of the "Operators applicable on more than two Scalar Values or Data Set Components", (see the section "Typical behaviours of the ML Operators").

Examples

Given the Data_ Set DS_1:

DS_1		
Id_1	**Id_2**	**Me_1**
1	A	"hello world"
2	A	"say hello"
3	A	"he"

4	A	"hello!"

Example 1: DS_r := replace (ds_1,"ello","i") results in:

DS_r		
Id_1	**Id_2**	**Me_1**
1	A	"hi world"
2	A	"say hi"
3	A	"he"
4	A	"hi! "

Example 2 (on component): DS_r := DS_1[calc Me_2:= replace (Me_1,"ello","i")] results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	" hello world"	"hi world"
2	A	" say hello"	"say hi"
3	A	"he"	"he"
4	A	"hello! "	"hi! "

String pattern location : instr

Syntax

instr (op**,** pattern**,** *start*, *occurrence* **)**

Input parameters

op the operand
pattern the string-pattern to be searched
start the position in the input string of the character from which the search starts
occurrence the occurrence of the pattern to search

Examples of valid syntaxes

instr (DS_1, “ab”, 2 , 3)
instr (DS_1, “ab”, 2)
instr (DS_1, “ab”, _ , 2)
instr (DS_1, “ab”)

Semantics for scalar operations

The operator returns the position in the input string of a specified string (pattern). The search starts from the startth character of the input string and finds the nthoccurrence of the pattern, returning the position of its first character.

- If start is omitted, the search starts from the 1st position.
- If nthoccurrence is omitted, the value is 1.

If the nthoccurrence of the string-pattern after the startth character is not found in the input string, the returned value is 0.

For example:

instr ("abcde", "c") gives 3
instr ("abcdecfrxcwsd", "c", _ , 3) gives 10
instr ("abcdecfrxcwsd", "c", 5 , 3) gives 0

Input parameters type

op :: dataset { measure<string> _ }
| component<string>
| string

pattern :: component<string>
| string

start :: component < integer [value >= 1] >
| integer [value >= 1]

occurrence :: component < integer [value >= 1] >
| integer [value >= 1]

Result type

result :: dataset { measure<integer[value >= 0]> int_var }
| component<integer[value >= 0]>
| integer[value >= 0]

Additional constraints

For operations at Data Set level, the input Data Set must have exactly one *string* type Measure.

Behaviour

As for the invocations at Data Set level, the operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component", as for the invocations at Component or Scalar level, the operator has the behaviour of the "Operators applicable on more than two Scalar Values or Data Set Components", (see the section "Typical behaviours of the ML Operators").

If op is a Data Set then **instr** returns a dataset with a single measure int_var of type *integer*.

Examples

Given the Data Set DS_1:

DS_1		
Id_1	**Id_2**	**Me_1**
1	A	"hello world"
2	A	"say hello"
3	A	"he"
4	A	"hi, hello! "

Example 1: DS_r:= instr(ds_1,"hello") results in

DS_r		
Id_1	**Id_2**	**int_var**
1	A	1
2	A	5
3	A	0
4	A	5

Example 2 (on component): DS_r := DS_1[calc Me_2:=instr(Me_1,"hello")] results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	"hello world"	1
2	A	"say hello"	5

3	A	"he"	0
4	A	"hi, hello!"	5

Given the Data Set DS_2:

DS_2			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	"hello"	"world"
2	B	NULL	"hi"

Example 3 (applying the **instr** *operator at component level to a multi Measure Data Set):*

DS_r := DS_2 [calc Me_10:= instr(Me_1, "o"), Me_20:=instr(Me_2, "o")]   results in:

DS_r					
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_10**	**Me_20**
1	A	"hello"	"world"	5	2
2	B	NULL	"hi"	null	0

Example 4 (applying the instr operator at Data Set level to a multi Measure Data Set):

DS_r := instr(DS_2, "o")   would give error because DS_2 has more Measures.

String length :   **length**

Syntax

length (op **)**

Input Parameters

op   the operand

Examples of valid syntaxes

length("Hello, World!")
length(DS_1)

Semantics for scalar operations

Returns the length of a string. For example, length("Hello, World!") gives 13
For the empty string "" the value 0 is returned

Input Parameters type

op ::   dataset { measure<string> _ }
| component<string>
| string

Result type

result ::   dataset { measure<integer[value >= 0]> int_var }
| component<integer[value >= 0]>
| integer[value >= 0]

Additional constraints

For operations at Data Set level, the input Data Set must have exactly one *string* type Measure.

Behaviour

The operator has the behaviour of the "Operators changing the data type" (see the section "Typical behaviours of the ML Operators").
If op is a Data Set then **length** returns a dataset with a single measure int_var of type *integer*.

Examples

Given the Data Set DS_1

DS_1		
Id_1	**Id_2**	**Me_1**
1	A	"hello"
2	B	null

Example 1: DS_r := length(DS_1) results in:

DS_r		
Id_1	**Id_2**	**int_var**
1	A	5
2	B	null

Example 2 (on component): DS_r:= DS_1[calc Me_2:=length(Me_1)] results in

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	"hello"	5
2	B	null	null

Given the Data Set DS_2:

DS_2			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	A	"hello"	"world"
2	B	null	"hi"

Example 3 (applying the **length** *operator at component level to a multi Measure Data Set)*:

DS_r := DS_2 [calc Me_10:= length(Me_1), Me_20:=length(Me_2)] results in:

DS_r					
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_10**	**Me_20**
1	A	"hello"	"world"	5	5
2	B	null	"hi"	null	2

*Example 4 (**length** operator applied at Data Set level to a multi Measure Data Set):*

DS_r := length(DS_2) would give error because DS_2 has more Measures.

VTL-ML - Numeric operators

Unary plus : +

Syntax

+ op

Input parameters

op the operand

Examples of valid syntaxes

+ DS_1

+ 3

Semantics for scalar operations

The operator **+** returns the operand unchanged. For example:

+ 3 gives 3

+ (- 5) gives - 5

Input Parameters type

op :: dataset { measure<number> _+ }
| component<number>
| number

Result type

result :: dataset { measure<number> _+ }
| component<number>
| number

Additional constraints

None.

Behaviour

The operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component" (see the section "Typical behaviours of the ML Operators").
According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of the operand is *integer* then the result has type *integer*. If the type of the operand is *number* then the result has type *number*.

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	Id_2	Me_1	Me_2
10	A	1.0	5
10	B	2.3	10
11	A	3.2	12

Example 1: DS_r := + DS_1 results in:

DS_r			
Id_1	Id_2	Me_1	Me_2
10	A	1.0	5

10	B	2.3	10
11	A	3.2	12

Example 2 (on components): DS_r := DS_1 [calc Me_3 := + Me_1] results in:

DS_r				
Id_1	Id_2	Me_1	Me_2	Me_3
10	A	1.0	5	1.0
10	B	2.3	10	2.3
11	A	3.2	12	3.2

Unary minus: -

Syntax

- op

Input parameters

op the operand

Examples of valid syntaxes

- DS_1
- 3

Semantics for scalar operations

The operator - inverts the sign of op. For example:

- 3 gives - 3
- (- 5) gives 5

Input Parameters type

op :: dataset { measure<number> _+ }
| component<number>
| number

Result type

result :: dataset { measure<number> _+ }
| component<number>
| number

Additional constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”).
According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of the operand is *integer* then the result has type *integer*. If the type of the operand is *number* then the result has type *number*.

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	1	5.0

10	B	2	10.0
11	A	3	12.0

Example 1: DS_r := - DS_1 results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	-1	-5.0
10	B	-2	-10.0
11	A	-3	-12.0

Example 2 (on components): DS_r := DS_1 [calc Me_3 := - Me_1] results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_3**
10	A	1	5.0	-1
10	B	2	10.0	-2
11	A	3	12.0	-3

Addition : **+**

Syntax

op1 **+** op2

Input parameters

op1 the first addendum

op2 the second addendum

Examples of valid syntaxes

DS_1 + DS_2

3 + 5

Semantics for scalar operations

The operator addition returns the sum of two numbers. For example:

3 + 5 gives 8

Input parameters type

op1, op2 :: dataset { measure<number> _+ }
| component<number>
| number

Result type

result :: dataset { measure<number> _+ }
| component<number>
| number

Additional constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components” (see the section “Typical behaviours of the ML Operators”).
According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	5	5.0
10	B	2	10.5
11	A	3	12.2
11	B	4	20.3

DS_2			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	10	3.0
10	C	11	6.2
11	B	6	7.0

Example 1: DS_r := DS_1 + DS_2 results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	15	8.0
11	B	10	27.3

Example 2: DS_r := DS_1 + 3 results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	8	8.0
10	B	5	13.5
11	A	6	15.2
11	B	7	23.3

Example 3 (on components): DS_r := DS_1 [calc Me_3 := Me_1 + 3.0] results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_3**
10	A	5	5.0	8.0
10	B	2	10.5	5.0
11	A	3	12.2	6.0
11	B	4	20.3	7.0

Subtraction : -

Syntax

op1 **-** op2

Input Parameters

op1 the minuend
op2 the subtrahend

Examples of valid syntaxes

DS_1 - DS_2
3 - 5

Semantics for scalar operations

The operator subtraction returns the difference of two numbers. For example:

3 - 5 gives - 2

Input Parameters type

op1, op2:: dataset { measure<number> _+ }
| component<number>
| number

Result type

result :: dataset { measure<number> _+ }
| component<number>
| number

Additional constraints

None.

Behaviour

The operator has the behaviour of the "Operators applicable on two Scalar Values or Data Sets or Data Set Components" (see the section "Typical behaviours of the ML Operators").
According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	5	5.0
10	B	2	10.5
11	A	3	12.2
11	B	4	20.3

DS_2			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	10	3.0
10	C	11	6.2
11	B	6	7.0

Example 1: DS_r := DS_1 - DS_2 results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	-5	2.0
11	B	-2	13.3

Example 2: DS_r := DS_1 - 3 results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	2	2.0
10	B	-1	7.5
11	A	0	9.2
11	B	1	17.3

Example 3 (on components): DS_r := DS_1 [calc Me_3 := Me_1 - 3] results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_3**
10	A	5	5.0	2
10	B	2	10.5	-1
11	A	3	12.2	0
11	B	4	20.3	1

Multiplication : *

Syntax

op1 * op2

Input parameters

op1 the multiplicand
op2 the multiplier

Examples of valid syntaxes

DS_1 * DS_2
3 * 5

Semantics for scalar operations

The operator multiplication returns the product of two numbers. For example:

3 * 5 gives 15

Input parameters type

op1, op2 :: dataset { measure<number> _+ }
| component<number>
| number

Result type

result :: dataset { measure<number> _+ }
| component<number>
| number

Additional constraints

None.

Behaviour

The operator has the behaviour of the "Operators applicable on two Scalar Values or Data Sets or Data Set Components" (see the section "Typical behaviours of the ML Operators").
According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	100	7.6
10	B	10	12.3
11	A	20	25.0
11	B	2	20.0

DS_2			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	1	2.0
10	C	5	3.0
11	B	2	1.0

Example 1: DS_r := DS_1 * DS_2 results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	100	15.2
11	B	4	20.0

Example 2: DS_r := DS_1 * -3 results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	-300	-22.8
10	B	-30	-36.9
11	A	-60	-75.0
11	B	-6	-60.0

Example 3 (on components): DS_r := DS_1 [calc Me_3 := Me_1 * Me_2] results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_3**
10	A	100	7.6	760.0
10	B	10	12.3	123.0
11	A	20	25.0	500.0
11	B	2	20.0	40.0

Division : /

Syntax

op1 **/** op2

Input parameters

op1 the dividend
op2 the divisor

Examples of valid syntaxes

DS_1 / DS_2
3 / 5

Semantics for scalar operations

The operator division divides two numbers. For example:

3 / 5 gives 0.6

Input parameters type

op1, op2 :: dataset { measure<number> _+ }
| component<number>
| number

Result type

result :: dataset { measure<number> _+ }
| component<number>
| number

Additional constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components” (see the section “Typical behaviours of the ML Operators”).
According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. The result has type *number*.
If op2 is 0 then the operation generates a run-time error.

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	100	7.6
10	B	10	12.3

11	A	20	25.0
11	B	10	12.3

DS_2			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	1	2.0
10	C	5	3.0
11	B	2	1.0

Example 1: DS_r := DS_1 / DS_2 results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	100	3.8
11	B	10	25.0

Example 2: DS_r := DS_1 / 10 results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	10	0.76
10	B	1	1.23
11	A	2	2.5
11	B	0.2	2.0

Example 3 (on components): DS_r := DS_1 [calc Me_3 := Me_2 / Me_1] results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_3**
10	A	100	7.6	0.076
10	B	10	12.3	1.23
11	A	20	25.0	1.25
11	B	2	20.0	10.0

Modulo : mod

Syntax

mod (op1 **,** op2 **)**

Input parameters

op1 the dividend
op2 the divisor

Examples of valid syntaxes

mod (DS_1, DS_2)
mod (DS_1, 5)
mod (5, DS_2)
mod (5, 2)

Semantics for scalar operations

The operator **mod** returns the remainder of op1 divided by op2. It returns op1 if divisor op2 is 0. For example:

mod (5, 2) gives 1
mod (5, -2) gives -1
mod (8, 2) gives 0
mod (9, 0) gives 9

Input Parameters type

op1, op2 :: dataset { measure<number> _+ }
| component<number>
| number
divisor :: number

Result type

result :: dataset { measure<number> _+ }
| component<number>
| number

Additional constraints

None.

Behaviour

The operator has the behaviour of the "Operators applicable on two Scalar Values or Data Sets or Data Set Components" (see the section "Typical behaviours of the ML Operators").
According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	100	0.7545
10	B	10	18.45
11	A	20	1.87
11	B	9	12.3

DS_2			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	1	0.25
10	C	5	3.0
11	B	2	2.0

Example 1: DS_r := mod (DS_1, DS_2) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	0	0.0045
11	B	1	0.3

Example 2: DS_r := mod (DS_1, 15) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	10	0.7545
10	B	10	3.45
11	A	5	1.87
11	B	9	12.3

Example 3 (on components): DS_r := DS_1[calc Me_3 := mod(DS_1#Me_1, 3.0)] results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**ME_3**
10	A	100	0.7545	1.0
10	B	10	18.45	1.0
11	A	20	1.87	2.0
11	B	9	12.3	0.0

Rounding : round

Syntax

round (op **,** *numDigit* **)**

Input parameters

op the operand
numDigit the number of positions to round to

Examples of valid syntaxes

round (DS_1 , 2)
round (DS_2)
round (3.14159 , 2)
round (3.14159 , _)

Semantics for scalar operations

The operator **round** rounds the operand to a number of positions at the right of the decimal point equal to the numDigit parameter. The decimal point is assumed to be at position 0. If numDigit is negative, the rouding happens at the left of the decimal point. The rounding operation leaves the numDigit position unchanged if the numDigit+1 position is between 0 and 4, otherwise it adds 1 to the number that is in the numDigit position. All the positions greater than numDigit are set to 0. The basic scalar type of the result is *integer* if numDigit is omitted, *number* otherwise.

For example:

round (3.14159, 2) gives 3.14
round (3.14159, 4) gives 3.1416
round (12345.6, 0) gives 12346.0

round (12345.6) gives 12346
round (12345.6, _) gives 12346
round (12345.6, -1) gives 12350.0

Input parameters type

op1 :: dataset { measure<number> _+ }
| component<number>
| number
numDigit:: component < integer >
| integer

Result type

result :: dataset { measure<number> _+ }
| component<number>
| number

Additional constraints

None.

Behaviour

As for the invocations at Data Set level, the operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component", as for the invocations at Component or Scalar level, the operator has the behaviour of the "Operators applicable on two Scalar Values or Data Sets or Data Set Components", (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_1**	**Me_1**	**Me_2**
10	A	7.5	5.9
10	B	7.1	5.5
11	A	36.2	17.7
11	B	44.5	24.3

Example 1: DS_r := round(DS_1, 0) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	8.0	6.0
10	B	7.0	6.0
11	A	36.0	18.0
11	B	45.0	24.0

Example 2 (on components): DS_r := DS_1 [calc Me_10:= round(Me_1)] results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_10**
10	A	7.5	5.9	8
10	B	7.1	5.5	7
11	A	36.2	17.7	36

11	B	44.5	24.3	45

Example 3 (on components) : DS_r := DS_1 [calc Me_20:= round(Me_1 , -1)] results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_20**
10	A	7.5	5.9	10
10	B	7.1	5.5	10
11	A	36.2	17.7	40
11	B	44.5	24.3	40

Truncation : trunc

Syntax

trunc (op **,** *numDigit* **)**

Input Parameters

op the operand

numDigit the number of position from which to trunc

Examples of valid syntaxes

trunc (DS_1 , 2)
trunc (DS_1)
trunc (3.14159 , 2)
trunc (3.14159 , _)

Semantics for scalar operations

The operator **trunc** truncates the operand to a number of positions at the right of the decimal point equal to the numDigit parameter. The decimal point is assumed to be at position 0. If numDigit is negative, the truncation happens at the left of the decimal point. The truncation operation leaves the numDigit position unchanged. All the positions greater than numDigit are eliminated. The basic scalar type of the result is *integer* if numDigit is omitted, *number* otherwise.

For example:

trunc (3.14159, 2) gives 3.14
trunc (3.14159, 4) gives 3.1415
trunc (12345.6, 0) gives 12345.0
trunc (12345.6) gives 12345
trunc (12345.6, _) gives 12345
trunc(12345.6, -1) gives 12340.0

Input parameters type

op :: dataset { measure<number> _+ }
| component<number>
| number

numDigit :: component < integer >
| integer

Result type

result :: dataset { measure<number> _+ }
| component<number>
| number

Additional constraints

None.

Behaviour

As for the invocations at Data Set level, the operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component", as for the invocations at Component or Scalar level, the operator has the behaviour of the "Operators applicable on two Scalar Values or Data Sets or Data Set Components", (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_1**	**Me_1**	**Me_2**
10	A	7.5	5.9
10	B	7.1	5.5
11	A	36.2	17.7
11	B	44.5	24.3

Example 1: DS_r := trunc(DS_1, 0) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	7.0	5.0
10	B	7.0	5.0
11	A	36.0	17.0
11	B	44.0	24.0

Example 2 (on components): DS_r := DS_1[calc Me_10:= trunc(Me_1)] results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_10**
10	A	7.5	5.9	7
10	B	7.1	5.5	7
11	A	36.2	17.7	36
11	B	44.5	24.3	44

Example 3 (on components): DS_r := DS_1[calc Me_20:= trunc(Me_1 , -1)] results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_20**
10	A	7.5	5.9	0
10	B	7.1	5.5	0
11	A	36.2	17.7	30
11	B	44.5	24.3	40

Ceiling : ceil

Syntax

ceil (op **)**

Input parameters

op the operand

Examples of valid syntaxes

ceil (DS_1)
ceil (3.14159)

Semantics for scalar operations

The operator **ceil** returns the smallest integer greater than or equal to op.
For example:

ceil(3.14159) gives 4
ceil(15) gives 15
ceil(-3.1415) gives -3
ceil(-0.1415) gives 0

Input parameters type

op :: dataset { measure<number> _+ }
| component<number>
| number

Result type

result :: dataset { measure<integer> _+ }
| component< integer >
| integer

Additional constraints

None.

Behaviour

The operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component" (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_1**	**Me_1**	**Me_2**
10	A	7.0	5.9
10	B	0.1	-5.0
11	A	-32.2	17.7
11	B	44.5	-0.3

Example 1: DS_r := ceil (DS_1) results in:

DS_r			
Id_1	**Id_1**	**Me_1**	**Me_2**
10	A	7	6
10	B	1	-5
11	A	-32	18

11	B	45	0

Example 2 (on components): DS_r := DS_1 [Me_10 := ceil (Me_1)] results in:

DS_r				
Id_1	**Id_1**	**Me_1**	**Me_2**	**Me_10**
10	A	7.0	5.9	7
10	B	0.1	-5.0	1
11	A	-32.2	17.7	-32
11	B	44.5	-0.3	45

Floor: floor

Syntax

floor (op **)**

Input parameters

op the operand

Examples of valid syntaxes

floor (DS_1)
floor (3.14159)

Semantics for scalar operations

The operator **floor** returns the greatest integer which is smaller than or equal to op.
For example:

floor(3.1415) gives 3
floor(15) gives 15
floor(-3.1415) gives -4
floor(-0.1415) gives -1

Input parameters type

op :: dataset { measure<number> _+ }
| component<number>
| number

Result type

result :: dataset { measure<integer> _+ }
| component< integer >
| integer

Additional constraints

None.

Behaviour

The operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component" (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_1**	**Me_1**	**Me_2**

10	A	7.0	5.9
10	B	0.1	-5.0
11	A	-32.2	17.7
11	B	44.5	-0.3

Example 1: DS_r := floor (DS_1) results in:

DS_r			
Id_1	**Id_1**	**Me_1**	**Me_2**
10	A	7	5
10	B	0	-5
11	A	-33	17
11	B	44	-1

Example 2 (on components): DS_r := DS_1 [Me_10 := floor (Me_1)] results in:

DS_r				
Id_1	**Id_1**	**Me_1**	**Me_2**	**Me_10**
10	A	7.5	5.9	7
10	B	0.1	-5.5	0
11	A	-32.2	17.7	-33
11	B	44.5	-0.3	44

Absolute value : abs

Syntax

abs (op **)**

Input parameters

op the operand

Examples of valid syntaxes

abs (DS_1)
abs (-5)

Semantics for scalar operations

The operator **abs** calculates the absolute value of a number.
For example:

abs (-5.49) gives 5.49
abs (5.49) gives 5.49

Input parameters type

op :: dataset { measure<number> _+ }
| component<number>
| number

Result type

result :: dataset { measure<number [value >= 0]> _+ }
| component<number [value >= 0]>

| number [value >= 0]

Additional constraints

None.

Behaviour

The operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component" (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	0.484183	0.7545
10	B	-0.515817	-13.45
11	A	-1.000000	187.0

Example 1: DS_r := abs (DS_1) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	0.484183	0.7545
10	B	0.515817	13.45
11	A	1.000000	187

Example 2 (on components): DS_r := DS_1 [Me_10 := abs(Me_1)] results in:

DS_r				
Id_1	**Id_2**	**Me_1**	**Me_2**	**Me_10**
10	A	0.484183	0.7545	0.484183
10	B	-0.515817	-13.45	0.515817
11	A	-1.000000	187	1.000000

Exponential : exp

Syntax

exp (op **)**

Input parameters

op the operand

Examples of valid syntaxes

exp (DS_1)
exp (5)

Semantics for scalar operations

The operator **exp** returns e (base of the natural logarithm) raised to the op-th power.
For example;
exp (5) gives 148.41315…
exp (1) gives 2.71828… (the number e)

exp (0) gives 1.0
exp (-1) gives 0.36787… (the number 1/e)

Input parameters type

op:: dataset { measure<number> _+ }
| component<number>
| number

Result type

result :: dataset { measure<number[value > 0]> _+ }
| component<number [value > 0]>
| number[value > 0]

Additional constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	5	0.7545
10	B	8	13.45
11	A	2	1.87

Example 1: DS_r := exp(DS_1) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	148.413	2.126547
10	B	2980.95	693842.3
11	A	7.38905	6.488296

Example 2 (on components): DS_r := DS_1 [Me_1 := exp (Me_1)] results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	148.413	0.7545
10	B	2980.95	13.45
11	A	7.389	1.87

Natural logarithm : ln

Syntax

ln (op **)**

Input parameters

op the operand

Examples of valid syntaxes

ln (DS_1)

ln (148)

Semantics for scalar operations

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

For example:

ln (148) gives 4.997…

ln (e) gives 1.0

ln (1) gives 0.0

ln (0,5) gives -0.693…

Input parameters type

op :: dataset { measure<number [value > 0] > _+ }

| component<number [value > 0] >

| number [value > 0]

Result type

result :: dataset { measure<number > _+ }

| component<number >

| number

Additional constraints

None.

Behaviour

The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set Component” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	148.413	0.7545
10	B	2980.95	13.45
11	A	7.38905	1.87

Example 1: DS_r := ln(DS_1) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	5.0	-0.281700
10	B	8.0	2.598979
11	A	2.0	0.625938

Example 2 (on components): DS_r := DS_1 [Me_2 := ln (DS_1#Me_1) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	148.413	5.0
10	B	2980.95	8.0
11	A	7.38905	2.0

Power : power

Syntax

power (base **,** exponent **)**

Input parameters

base the operand
exponent the exponent of the power

Examples of valid syntaxes

power (DS_1, 2)
power (5, 2)

Semantics for scalar operations

The operator **power** raises a number (the base) to another one (the exponent).
For example:

power (5, 2) gives 25
power (5, 1) gives 5
power (5, 0) gives 1
power (5, -1) gives 0.2
power (-5, 3) gives -125

Input parameters type

base :: dataset { measure<number> _+ }
| component<number>
| number
exponent :: component<number>
| number

Result type

result :: dataset { measure<number> _+ }
| component<number>
| number

Additional constraints

None.

Behaviour

As for the invocations at Data Set level, the operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component", as for the invocations at Component or Scalar level, the operator has the behaviour of the "Operators applicable on two Scalar Values or Data Sets or Data Set Components", (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**

10	A	3	0.7545
10	B	4	13.45
11	A	5	1.87

Example 1: DS_r := power(DS_1, 2) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	9	0.56927
10	B	16	180.9025
11	A	25	3.4969

Example 2 (on components): DS_r := DS_1[calc Me_1 := power(Me_1, 2)] results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	9	0.7545
10	B	16	13.45
11	A	25	1.87

Logarithm : log

Syntax

log (op , num **)**

Input parameters

op the base of the logarithm
num the number to which the logarithm is applied

Examples of valid syntaxes

log (DS_1, 2)
log (1024, 2)

Semantics for scalar operations

The operator **log** calculates the logarithm of num base op.
For example:

log (1024, 2) gives 10
log (1024, 10) gives 3.01

Input parameters type

op :: dataset { measure<number [value > 1] > _+ }
| component<number [value > 1] >
| number [value > 1]
num :: component<integer [value > 0]>
| integer [value > 0]

Result type

result :: dataset { measure<number> _+ }
| component<number>
| number

Additional constraints

None.

Behaviour

As for the invocations at Data Set level, the operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component", as for the invocations at Component or Scalar level, the operator has the behaviour of the "Operators applicable on two Scalar Values or Data Sets or Data Set Components", (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	1024	0.7545
10	B	64	13.45
11	A	32	1.87

Example 1: DS_r := log (DS_1, 2) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	10.0	-0.40641
10	B	6.0	3.749534
11	A	5.0	0.903038

Example 2 (on components): DS_r := DS_1 [calc Me_1 := log (Me_1, 2)] results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	10.0	0.7545
10	B	6.0	13.45
11	A	5.0	1.87

Square root : **sqrt**

Syntax

sqrt (op **)**

Input parameters

op the operand

Examples of valid syntaxes

sqrt (DS_1)
sqrt (5)

Semantics for scalar operations

The operator **sqrt** calculates the square root of a number. For example:

sqrt (25) gives 5

Input parameters type

op :: dataset { measure<number [value >= 0] > _+ }
| component<number [value >= 0] >
| number [value >= 0]

Result type

result :: dataset { measure<number[value >= 0] > _+ }
| component<number[value >= 0] >
| number[value >= 0]

Additional constraints

None.

Behaviour

The operator has the behaviour of the "Operators applicable on one Scalar Value or Data Set or Data Set Component" (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	16	0.7545
10	B	81	13.45
11	A	64	1.87

Example 1: DS_r := sqrt(DS_1) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	4	0.86862
10	B	9	3.667424
11	A	8	1.367479

Example 2 (on components): DS_r := DS_1 [calc Me_1 := sqrt (Me_1)] results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
10	A	4	0.7545
10	B	9	13.45
11	A	8	1.87

VTL-ML - Comparison operators

Equal to : =

Syntax

left = right

Input parameters

left the left operand
right the right operand

Examples of valid syntaxes

DS_1 = DS_2

Semantics for scalar operations

The operator returns TRUE if the left is equal to right, FALSE otherwise.
For example:

5 = 9 gives: FALSE
5 = 5 gives: TRUE
“hello” = “hi” gives: FALSE

Input parameters type

left,
right :: dataset {measure<scalar> _ }
| component<scalar>
| scalar

Result type

result :: dataset { measure<boolean> bool_var }
| component<boolean>
| boolean

Additional constraints

Operands left and right must be of the same scalar type

Behaviour

The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	NULL
2012	G	Total	Total	0.286
2012	S	Total	Total	0.064
2012	M	Total	Total	0.043
2012	F	Total	Total	0.08
2012	W	Total	Total	0.08

Example 1: DS_r := DS_1 = 0.08 results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**bool_var**
2012	B	Total	Total	NULL
2012	G	Total	Total	FALSE
2012	S	Total	Total	FALSE
2012	M	Total	Total	FALSE
2012	F	Total	Total	TRUE
2012	W	Total	Total	TRUE

Example 2 (on Components): DS_r := DS_1 [calc Me_2 := Me_1 = 0.08] results in:

DS_r					
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**	**Me_2**
2012	B	Total	Total	NULL	NULL
2012	G	Total	Total	0.286	FALSE
2012	S	Total	Total	0.064	FALSE
2012	M	Total	Total	0.043	FALSE
2012	F	Total	Total	0.08	TRUE
2012	W	Total	Total	0.08	TRUE

Not equal to : < >

Syntax

left **<>** right

Input parameters

left the left operand
right the right operand

Examples of valid syntaxes

DS_1 <> DS_2

Semantics for scalar operations

The operator returns FALSE if the left is equal to right, TRUE otherwise.
For example:

5 <> 9 gives: TRUE
5 <> 5 gives: FALSE
“hello” <> “hi” gives: TRUE

Input parameters type

left,
right :: dataset {measure<scalar> _ }
| component<scalar>
| scalar

Result type

result :: dataset { measure<boolean> bool_var }
| component<boolean>
| boolean

Additional constraints

Operands left and right must be of the same scalar type

Behaviour

The operator has the typical behaviour of the "Operators changing the data type" (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
G	Total	Percentage	Total	7.1
R	Total	Percentage	Total	NULL

DS_2				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
G	Total	Percentage	Total	7.5
R	Total	Percentage	Total	3

Example 1: DS_r := DS_1 <> DS_2 results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**bool_var**
G	Total	Percentage	Total	TRUE
R	Total	Percentage	Total	NULL

Note that due to the behaviour for NULL values, if the value for Greece in the second operand had also been NULL, then the result would still be NULL for Greece.

Example 2 (on Components): DS_r := DS_1 [Me_2 := Me_1<>7.5] results in:

DS_r					
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**	**Me_2**
G	Total	Percentage	Total	7.5	TRUE
R	Total	Percentage	Total	3	NULL

Greater than : > >=

Syntax

left { > | >= }[1] right

Input parameters

left the left operand part of the comparison
right the right operand part of the comparison

Examples of valid syntaxes

DS_1 > DS_2
DS_1 >= DS_2

Semantics for scalar operations

The operator **>** returns TRUE if left is greater than right, FALSE otherwise.
The operator **>=** returns TRUE if left is greater than or equal to right, FALSE otherwise.
For example:

5 > 9	gives: FALSE
5 >= 5	gives: TRUE
"hello" > "hi"	gives: FALSE

Input parameters type

left,
right :: dataset {measure<scalar> _ }
| component<scalar>
| scalar

Result type

result :: dataset { measure<boolean> bool_var }
| component<boolean>
| boolean

Additional constraints

Operands left and right must be of the same scalar type

Behaviour

The operator has the typical behaviour of the "Operators changing the data type" (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Set DS_1:

DS_1					
Id_1	**Id_2**	**Id_3**	**Id_4**	**Id_5**	**Me_1**
2	G	2011	Total	Percentage	NULL
2	R	2011	Total	Percentage	12.2
2	F	2011	Total	Percentage	29.5

Example 1: DS_r := DS_1 > 20 results in:

DS_r					
Id_1	**Id_2**	**Id_3**	**Id_4**	**Id_5**	**bool_var**
2	G	2011	Total	Percentage	NULL
2	R	2011	Total	Percentage	FALSE
2	F	2011	Total	Percentage	TRUE

Example 2 (on Components): DS_r := DS_1 [Me_2 := Me_1 > 20] results in:

DS_r						
Id_1	**Id_2**	**Id_3**	**Id_4**	**Id_5**	**Me_1**	**Me_2**
2	G	2011	Total	Percentage	NULL	NULL
2	R	2011	Total	Percentage	12.2	FALSE
2	F	2011	Total	Percentage	29.5	TRUE

Given the left operand Data Set:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
G	Total	Percentage	Total	7.1
R	Total	Percentage	Total	42.5

and the right operand Data Set:

DS_2				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
G	Total	Percentage	Total	7.5
R	Total	Percentage	Total	33.7

Example 3: DS_r:= DS_1 > DS_2 results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**bool_var**
G	Total	Percentage	Total	FALSE
R	Total	Percentage	Total	TRUE

If the Me_1 column for Germany in the DS_2 Data Set had a NULL value the result would be:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**bool_var**
G	Total	Percentage	Total	NULL
R	Total	Percentage	Total	TRUE

Less than : < <=

Syntax

left { **<** | **<=** }[1] right

Input parameters

left the left operand
right the right operand

Examples of valid syntaxes

DS_1 < DS_2
DS_1 <= DS_2

Semantics for scalar operations

The operator **<** returns TRUE if left is smaller than right, FALSE otherwise.
The operator **<=** returns TRUE if left is smaller than or equal to right, FALSE otherwise.
For example:

5 < 4	gives: FALSE
5 <= 5	gives: TRUE
“hello” < “hi”	gives: TRUE

Input parameters type

left, right :: dataset {measure<scalar> _ }
| component<scalar>
| scalar

Result type

result :: dataset { measure<boolean> bool_var }
| component<boolean>
| boolean

Additional constraints

Operands left and right must be of the same scalar type

Behaviour

The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

Examples

Given the operand Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	11094850
2012	G	Total	Total	11123034
2012	S	Total	Total	46818219
2012	M	Total	Total	NULL
2012	F	Total	Total	5401267
2012	W	Total	Total	7954662

Example 1: DS_r := DS_1 < 15000000 results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**bool_var**
2012	B	Total	Total	TRUE
2012	G	Total	Total	TRUE
2012	S	Total	Total	FALSE
2012	M	Total	Total	NULL
2012	F	Total	Total	TRUE
2012	W	Total	Total	TRUE

Between : between

Syntax

between (op, from, to**)**

Input parameters

op   the Data Set to be checked
from   the left delimiter
to   the right delimiter

Examples of valid syntaxes

ds2 := between(ds1, 5,10)
ds2 := ds1 [calc m1 := between(me2, 5, 10)]

Semantics for scalar operations

The operator returns TRUE if op is greater than or equal to from and lower than or equal to to. In other terms, it is a shortcut for the following:

op >= from and op <= to

The types of op, from and to must be compatible scalar types.

Input parameters type

op ::   dataset {measure<scalar> _}
| component<scalar>
| scalar

from ::   scalar | component<scalar>
to ::   scalar | component<scalar>

Result type

result ::   dataset { measure<booelan> bool_var }
| component<boolean>
| boolean

Additional constraints

The type of the operand (i.e., the measure of the dataset, the type of the component, the scalar type) must be the same as that of from and to.

Behaviour

The operator has the typical behaviour of the "Operators changing the data type" (see the section "Typical behaviours of the ML Operators").

Examples

Given the following Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
G	Total	Percentage	Total	6
R	Total	Percentage	Total	-2

Example 1:   DS_r:= between(ds1, 5,10)   results in:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**bool_var**

G	Total	Percentage	Total	TRUE
R	Total	Percentage	Total	FALSE

Element of: in / not_in

Syntax

op **in** <u>collection</u>
op **not_in** <u>collection</u>

<u>collection</u> ::= set | valueDomainName

Input parameters

op	the operand to be tested
<u>collection</u>	the the Set or the Value Domain which contains the values
set	the Set which contains the values (it can be a Set name or a Set literal)
valueDomainName	the name of the Value Domain which contains the values

Examples of valid syntaxes

ds := ds_2 in {1,4,6} as usual, here the braces denote a set literal (it contains the values 1, 4 and 6)
ds := ds_3 in mySet
ds := ds_3 in myValueDomain

Semantics for scalar operations

The **in** operator returns TRUE if op belongs to the collection, FALSE otherwise.
The **not_in** operator returns FALSE if op belongs to the collection, TRUE otherwise.
For example:

1 in { 1, 2, 3 }	returns	TRUE
"a" in { "c, "ab", "bb", "bc" }	returns	FALSE
"b" not_in { "b", "hello", "c"}	returns	FALSE
"b" not_in { "a", "hello", "c"}	returns	TRUE

Input parameters type

op :: dataset {measure<scalar> _ }
| component<scalar>
| scalar
collection :: set<scalar> | name<value_domain>

Result type

result :: dataset { measure<boolean> bool_var }
| component<boolean>
| boolean

Additional constraints

The operand must be of a basic scalar data type compatible with the basic scalar type of the collection.

Behaviour

Semantics

The **in** operator evaluates to TRUE if the operand is an element of the specified collection and FALSE otherwise, the **not_in** the opposite.
The operator has the typical behaviour of the "Operators changing the data type" (see the section "Typical behaviours of the ML Operators").
The collection can be either a *set* of values defined in line or a name that references an externally defined Value Domain or Set.

Examples

Given the operand Data Set DS_1:

DS_1		
Id_1	**Id_2**	**Me_1**
2012	BS	0
2012	GZ	4
2012	SQ	9
2012	MO	6
2012	FJ	7
2012	CQ	2

Example 1:

DS_r := DS_1 in { “BS”, “MO”, “HH”, “PP” }   results in:

DS_r		
Id_1	**Id_2**	**bool_var**
2012	BS	TRUE
2012	GZ	FALSE
2012	SQ	FALSE
2012	MO	TRUE
2012	FJ	FALSE
2012	CQ	FALSE

Example 2 (on Components):

DS_r := DS_1 [calc Me_2:= Me_1 in { “BS”, “MO”, “HH”, “PP” }]   results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
2012	BS	0	TRUE
2012	GZ	4	FALSE
2012	SQ	9	FALSE
2012	MO	6	TRUE
2012	FJ	7	FALSE
2012	CQ	2	FALSE

Given the previos Data Set DS_1 and the following Value Domain named myGeoValueDomain (which has the basic scalar type *string*) *:*

myGeoValueDomain	
Code	***Meaning***
AF	Afghanistan
BS	Bahamas
FJ	Fiji
GA	Gabon
KH	Cambodia

MO	Macao
PK	Pakistan
QA	Quatar
UG	Uganda

Example 3 (on external Value Domain):

DS_r := DS_1#Id_2 in myGeoValueDomain results in:

DS_r		
Id_1	**Id_2**	**bool_var**
2012	BS	TRUE
2012	GZ	FALSE
2012	SQ	FALSE
2012	MO	TRUE
2012	FJ	TRUE
2012	CQ	FALSE

match_characters

match_characters

Syntax

match_characters (op **,** pattern **)**

Input parameters

op the dataset to be checked
pattern the regular expression to check the Data Set or the Component against

Examples of valid syntaxes

match_characters(ds1, “[abc]+\d\d”)
ds1 [**calc** m1 := match_characters(ds1, “[abc]+\d\d”)]

Semantics for scalar operations

match_characters returns TRUE if op matches the regular expression regexp, FALSE otherwise. The string regexp is an Extended Regular Expression as described in the POSIX standard. Different implementations of VTL may implement different versions of the POSIX standard therefore it is possible that **match_characters** may behave in slightly different ways.

Input parameters type

op :: dataset {measure<string> _}
| component<string>
| string
pattern :: string | component<string>

Result type

result :: dataset { measure<booelan> bool_var }

| component<boolean>
| boolean

Additional constraints

If op is a Data Set then it has exactly one measure.
pattern is a POSIX regular expression.

Behaviour

The operator has the typical behaviour of the "Operators changing the data type" (see the section "Typical behaviours of the ML Operators").

Examples

Given the following Dataset DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
G	Total	Percentage	Total	AX123
R	Total	Percentage	Total	AX2J5

DS_r:=(ds1, "[:alpha:]{2}[:digit:]{3}") results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**bool_var**
G	Total	Percentage	Total	TRUE
R	Total	Percentage	Total	FALSE

Isnull: isnull

Syntax

isnull (op **)**

Input parameters

operand mandatory the operand

Examples of valid syntaxes

isnull(DS_1)

Semantics for scalar operations

The operator returns TRUE if the value of the operand is NULL, FALSE otherwise.

Examples

isnull("Hello") gives: FALSE
isnull(NULL) gives: TRUE

Input parameters type

op :: dataset {measure<scalar> _}
| component<scalar>
| scalar

Result type

result :: dataset { measure<boolean> bool_var }
| component<boolean>
| boolean

Additional constraints

If op is a Data Set then it has exactly one measure.

Behaviour

The operator has the typical behaviour of the "Operators changing the data type" (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	11094850
2012	G	Total	Total	11123034
2012	S	Total	Total	NULL
2012	M	Total	Total	417546
2012	F	Total	Total	5401267
2012	N	Total	Total	NULL

Example 1: DS_r := isnull(DS_1) results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**bool_var**
2012	B	Total	Total	FALSE
2012	G	Total	Total	FALSE
2012	S	Total	Total	TRUE
2012	M	Total	Total	FALSE
2012	F	Total	Total	FALSE
2012	N	Total	Total	TRUE

Example 2 (on Components): DS_r := DS_1[Me_2 := is_null(Me_1)] results in:

DS_r					
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**	**Me_2**
2012	B	Total	Total	11094850	FALSE
2012	G	Total	Total	11123034	FALSE
2012	S	Total	Total	NULL	TRUE
2012	M	Total	Total	417546	FALSE
2012	F	Total	Total	5401267	FALSE
2012	N	Total	Total	NULL	TRUE

Exists in : exists_in

Syntax

exists_in (op1**,** op2 { **,** retain } **)**

retain ::= **true** | **false** | **all**

Input parameters

op1 the operand dataset
op2 the operand dataset
retain the optional parameter to specify the Data Points to be returned (default: **all**)

Examples of valid syntaxes

exists_in (DS_1, DS_2, true)
exists_in (DS_1, DS_2)
exists_in (DS_1, DS_2, all)

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op1,
op2 :: dataset

Result type

result :: dataset { measure<boolean> bool_var }

Additional constraints

op2 has all the identifier components of op1.

Behaviour

The operator checks if the combinations of values of the Identifiers existing in op1also exist in op2.
The result has the same Identifiers as op1 and a *boolean* Measure bool_var whose value, for each Data Point of op1, is TRUE if the combination of values of the Identifier Components existing in op1 is found in a Data Point of op2, FALSE otherwise. If retain is **all** then both the Data Points having bool_var = TRUE and bool_var = FALSE are returned.
If retain is **true** then only the data points with bool_var = TRUE are returned. If retain is **false** then only the Data Points with bool_var = FALSE are returned.If the retain parameter is omitted, the default is all.
The operator has the typical behaviour of the "Operators changing the data type" (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	11094850
2012	G	Total	Total	11123034
2012	S	Total	Total	46818219
2012	M	Total	Total	417546
2012	F	Total	Total	5401267
2012	W	Total	Total	7954662

DS_2				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	0.023
2012	G	Total	M	0.286
2012	S	Total	Total	0.064
2012	M	Total	M	0.043
2012	F	Total	Total	NULL
2012	W	Total	Total	0.08

Example 1: DS_r := exists_in (DS_1, DS_2, all) results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**bool_var**
2012	B	Total	Total	TRUE
2012	G	Total	Total	FALSE
2012	S	Total	Total	TRUE
2012	M	Total	Total	FALSE
2012	F	Total	Total	TRUE
2012	W	Total	Total	TRUE

Example 2: DS_r := exists_in (DS_1, DS_2, true) results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**bool_var**
2012	B	Total	Total	TRUE
2012	S	Total	Total	TRUE
2012	F	Total	Total	TRUE
2012	W	Total	Total	TRUE

Example 3: DS_r := exists_in (DS_1, DS_2, false) results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**bool_var**
2012	G	Total	Total	FALSE
2012	M	Total	Total	FALSE

VTL-ML - Boolean operators

Logical conjunction: and

Syntax

op1 **and** op2

Input parameters

op1 the first operand
op2 the seconf operand

Examples of valid syntaxes

DS_1 and DS_2

Semantics for scalar operations

The **and** operator returns TRUE if both operands are TRUE, otherwise FALSE. The two operands must be of *boolean* type.

For example:

FALSE **and** FALSE gives FALSE
FALSE **and** TRUE gives FALSE
TRUE **and** FALSE gives FALSE
TRUE **and** TRUE gives TRUE

Input parameters type

op1,
op2 :: dataset {measure<boolean> _ }
| component<boolean>
| boolean

Result type

result :: dataset { measure<boolean> _}
| component<boolean>
| boolean

Additional constraints

None.

Behaviour

The operator has the typical behaviour of the "Behaviour of Boolean operators" (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	TRUE
M	64	B	2013	FALSE
M	65	B	2013	TRUE
F	15	U	2013	FALSE
F	64	U	2013	FALSE
F	65	U	2013	TRUE

DS_2				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
M	15	B	2013	FALSE
M	64	B	2013	TRUE
M	65	B	2013	TRUE
F	15	U	2013	TRUE
F	64	U	2013	FALSE
F	65	U	2013	FALSE

Example 1: DS_r:= DS_1 and DS_2 results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
M	15	B	2013	FALSE
M	64	B	2013	FALSE
M	65	B	2013	TRUE
F	15	U	2013	FALSE
F	64	U	2013	FALSE
F	65	U	2013	FALSE

Example 2 (on Components): DS_r := DS_1 [Me_2:= Me_1 and true] results in:

DS_r					
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**	**Me_2**
M	15	B	2013	TRUE	TRUE
M	64	B	2013	FALSE	FALSE
M	65	B	2013	TRUE	TRUE
F	15	U	2013	FALSE	FALSE
F	64	U	2013	FALSE	FALSE
F	65	U	2013	TRUE	TRUE

Logical disjunction : or

Syntax

op1 **or** op2

Input parameters

op1 the first operand
op2 the second operand

Examples of valid syntaxes

DS_1 or DS_2

Semantics for scalar operations

The **or** operator returns TRUE if at least one of the operands is TRUE, otherwise FALSE. The two operands must be of *boolean* type.

For example:

FALSE or FALSE gives FALSE
FALSE or TRUE gives TRUE
TRUE or FALSE gives TRUE
TRUE or TRUE gives TRUE

Input parameters type

op1,
op2 :: dataset {measure<boolean> _ }
| component<boolean>
| boolean

Result type

result :: dataset { measure<boolean> _ }
| component<boolean>
| boolean

Additional constraints

None.

Behaviour

The operator has the typical behaviour of the "Behaviour of Boolean operators" (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
M	15	B	2013	TRUE
M	64	B	2013	FALSE
M	65	B	2013	TRUE
F	15	U	2013	FALSE
F	64	U	2013	FALSE
F	65	U	2013	TRUE

DS_2				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
M	15	B	2013	FALSE
M	64	B	2013	TRUE
M	65	B	2013	TRUE
F	15	U	2013	TRUE
F	64	U	2013	FALSE
F	65	U	2013	FALSE

Example 1: DS_r:= DS_1 or DS_2 results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**

M	15	B	2013	TRUE
M	64	B	2013	TRUE
M	65	B	2013	TRUE
F	15	U	2013	TRUE
F	64	U	2013	FALSE
F	65	U	2013	TRUE

Example 2 (on Components): DS_r:= DS_1 [Me_2:= Me_1 or true] results in:

DS_r					
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**	**Me_2**
M	15	B	2013	TRUE	TRUE
M	64	B	2013	FALSE	TRUE
M	65	B	2013	TRUE	TRUE
F	15	U	2013	FALSE	TRUE
F	64	U	2013	FALSE	TRUE
F	65	U	2013	TRUE	TRUE

Exclusive disjunction : xor

Syntax

op1 **xor** op2

Input parameters

op1 the first operand
op2 the second operand

Examples of valid syntaxes

DS_1 xor DS_2

Semantics for scalar operations

The **xor** operator returns TRUE if only one of the operand is TRUE (but not both), FALSE otherwise. The two operands must be of *boolean* type.
For example:

FALSE xor FALSE gives FALSE
FALSE xor TRUE gives TRUE
TRUE xor FALSE gives TRUE
TRUE xor TRUE gives FALSE

Input parameters type

op1,
op2 :: dataset {measure<boolean> _ }
| component<boolean>
| boolean

Result type

result :: dataset { measure<boolean> _ }
| component<boolean>
| boolean

Additional constraints

None.

Behaviour

The operator has the typical behaviour of the "Behaviour of Boolean operators" (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
M	15	B	2013	TRUE
M	64	B	2013	FALSE
M	65	B	2013	TRUE
F	15	U	2013	FALSE
F	64	U	2013	FALSE
F	65	U	2013	trTRUEue

DS_2				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
M	15	B	2013	FALSE
M	64	B	2013	TRUE
M	65	B	2013	TRUE
F	15	U	2013	TRUE
F	64	U	2013	FALSE
F	65	U	2013	FALSE

Example 1: DS_r:=DS_1 xor DS_2 results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
M	15	B	2013	TRUE
M	64	B	2013	TRUE
M	65	B	2013	FALSE
F	15	U	2013	TRUE
F	64	U	2013	FALSE
F	65	U	2013	TRUE

Example 2 (on Components): DS_r:= DS_1 [Me_2:= Me_1 xor true] results in:

DS_r					
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**	**Me_2**
M	15	B	2013	TRUE	FALSE
M	64	B	2013	FALSE	TRUE

M	65	B	2013	TRUE	FALSE
F	15	U	2013	FALSE	TRUE
F	64	U	2013	FALSE	TRUE
F	65	U	2013	TRUE	FALSE

Logical negation : **not**

Syntax

not op

Input parameters

op the operand

Examples of valid syntaxes

not DS_1

Semantics for scalar operations

The **not** operator returns TRUE if op is FALSE, otherwise TRUE. The input operand must be of *boolean* type.
For example:

not FALSE gives TRUE
not TRUE gives FALSE

Input parameters type

op :: dataset {measure<boolean> _ }
| component<boolean>
| boolean

Result type

result :: dataset { measure<boolean> _ }
| component<boolean>
| boolean

Additional constraints

None.

Behaviour

The operator has the typical behaviour of the "Behaviour of Boolean operators" (see the section "Typical behaviours of the ML Operators").

Examples

Given the operand Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
M	15	B	2013	TRUE
M	64	B	2013	FALSE
M	65	B	2013	TRUE
F	15	U	2013	FALSE
F	64	U	2013	FALSE
F	65	U	2013	TRUE

Example 1: DS_r:= not DS_1 results in:

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1
M	15	B	2013	FALSE
M	64	B	2013	TRUE
M	65	B	2013	FALSE
F	15	U	2013	TRUE
F	64	U	2013	TRUE
F	65	U	2013	false

Example 2 (on Components): DS_r:= DS_1 [calc Me_2 := not Me_1] results in:

DS_r

Id_1	Id_2	Id_3	Id_4	Me_1	Me_2
M	15	B	2013	TRUE	FALSE
M	64	B	2013	FALSE	TRUE
M	65	B	2013	TRUE	FALSE
F	15	U	2013	FALSE	TRUE
F	64	U	2013	FALSE	TRUE
F	65	U	2013	TRUE	FALSE

VTL-ML - Time operators

This chapter describes the **time** operators, which are the operators dealing with **time, date** and **time_period** basic scalar types. The general aspects of the behaviour of these operators is described in the section “Behaviour of the Time Operators”.

The *time* data type is the most general type and denotes a generic time interval, having start and end points in time and therefore a duration, which is the time intervening between the start and end points. The *date* data type denotes a generic time instant (a point in time), which is a time interval with zero duration. The *time_period* data type denotes a regular time interval whose regular duration is explicitly represented inside each time_period value and is named period_indicator. In some sense, we say that *date* and *time_period* are special cases of *time*, the former with coinciding extremes and zero duration and the latter with regular duration. The *time* data type is overarching in the sense that it comprises *date* and *time_period*. Finally, *duration* data type represents a generic time span, independently of any specific start and end date.

The time, date and time period formats used here are explained in the User Manual in the section “External representations and literals used in the VTL Manuals”.

The period indicator P id of the *duration* type and its possible values are:

D	Day
W	Week
M	Month
Q	Quarter
S	Semester
A	Year

As already said, these representation are not prescribed by VTL and are not part of the VTL standard, each VTL system can personalize the representation of time, date, time_period and duration as desired. The formats shown above are only the ones used in the examples.

For a fully-detailed explanation, please refer to the User Manual.

Period indicator : **period_indicator**

The operator **period_indicator** extracts the period indicator from a *time_period* value.

Syntax

period_indicator ({ *op* **})**

Input parameters

op the operand

Examples of valid syntaxes

period_indicator (ds_1)

period_indicator (if used in a clause the operand op can be omitted)

Semantics for scalar operations

period_indicator returns the period indicator of a *time_period* value. The period indicator is the part of the *time_period* value which denotes the duration of the time period (e.g. day, week, month …).

Input parameters type

op :: dataset { identifier <time_period> _ , identifier _* }
| component<time_period>
| time_period

Result type

result :: dataset { measure<duration> duration_var }
| component <duration>
| duration

Additional constraints

If op is a Data Set then it has exactly an Identifier of type *time_period* and may have other Identifiers. If the operator is used in a clause and op is omitted, then the Data Set to which the clause is applied has exactly an Identifier of type *time_period*.

Behaviour

The operator extracts the period indicator part of the *time_period* value. The period indicator is computed for each Data Point. When the operator is used in a clause, it extracts the period indicator from the *time_period* value the Data Set to which the clause is applied.
The operator returns a Data Set with the same Identifiers of op and one Measure of type *duration* named duration_var. As for all the Variables, a proper Value Domain must be defined to contain the possible values of the period indicator and duration_var. The values used in the examples are listed at the beginning of this chapter "VTL-ML Time operators".

Examples

Given the Data Set DS_1:

DS_r			
Id_1	**Id_2**	**Id_3**	**Me_1**
A	1	2010	10
A	1	2013Q1	50

Example 1: DS_r := period_indicator (DS_1) results in:

DS_r			
Id_1	**Id_2**	**Id_3**	**duration_var**
A	1	2010	A
A	1	2013Q1	Q

Example 2 (on component): DS_r := DS_1 [filter period_indicator (Id_3) = "A"] results in:

DS_r			
Id_1	**Id_2**	**Id_3**	**Me_1**
A	1	2010	10

Fill time series : fill_time_series

Syntax

fill_time_series (**op { **, limitsMethod } **)**

limitsMethod ::= **single** | **all**

Input parameters

op the operand
limitsMethod method for determining the limits of the time interval to be filled (default: **all**)

Examples of valid syntaxes

fill_time_series (ds)
fill_time_series (ds, all)

Semantics for scalar operations

The fill_time_series operator does not perform scalar operations.

Input parameters type:

op :: dataset { identifier <time > _ , identifier _* }

Result type:

result :: dataset { identifier <time > _ , identifier _* }

Additional constraints

The operand op has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.

Behaviour

This operator can be applied only on Data Sets of time series and returns a Data Set of time series.
The operator fills the possibly missing Data Points of all the time series belonging to the operand op within the time limits automatically determined by applying the limit_method.
If limitsMmethod is **all**, the time limits are determined with reference to all the time_series of the Data Set: the limits are the minimum and the maximum values of the reference time Identifier Component of the Data Set.
If limitsMmethod is **single**, the time limits are determined with reference to each single time_series of the Data Set: the limits are the minimum and the maximum values of the reference time Identifier Component of the time series.
The expected Data Points are determined, for each time series, by considering the limits above and the *period* (*frequency*) of the time series: all the Identifiers are kept unchanged except the reference time Identifier, which is increased of one *period* at a time (e.g. day, week, month, quarter, year) from the lower to the upper time limit. For each increase, an expected Data Point is identified.
If this expected Data Points is missing, it is added to the Data Set. For the added Data Points, Measures and Attributes assume the NULL value.
The output Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set. The ouput Data Set contains the same time series as the operand, because the time series Identifiers (all the Identifiers except the reference time Identifier) are not changed.
As mentioned in the section "Behaviour of the Time Operators", the operator is assumed to know which is the reference time Identifier as well as the *period* of each time series.

Examples

Given the Data Set DS_1, which contains *yearly* time series, where Id_2 is the reference time Identifier of *time* type:

DS_1		
Id_1	**Id_2**	**Me_1**
A	2010-01/2010-12	"hello world"
A	2012-01/2012-12	"say hello"
A	2013-01/2013-12	"he"
B	2011-01/2011-12	"hi, hello! "
B	2012-01/2012-12	"hi"
B	2014-01/2014-12	"hello!"

Example 1: DS_r := fill_time_series (DS_1, single) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010-01/2010-12	"hello world"

A	2011-01/2011-12	NULL
A	2012-01/2012-12	"say hello"
A	2013-01/2013-12	"he"
B	2011-01/2011-12	"hi, hello! "
B	2012-01/2012-12	"hi”
B	2013-01/2013-12	NULL
B	2014-01/2014-12	“hello!”

Example 2: DS_r := fill_time_series (DS_1, all) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010-01/2010-12	"hello world"
A	2011-01/2011-12	NULL
A	2012-01/2012-12	"say hello"
A	2013-01/2013-12	"he"
A	2014-01/2014-12	NULL
B	2010-01/2010-12	NULL
B	2011-01/2011-12	"hi, hello! "
B	2012-01/2012-12	"hi”
B	2013-01/2013-12	NULL
B	2014-01/2014-12	“hello!”

Given the Data Set DS_2, which contains *yearly* time series, where Id_2 is the reference time Identifier of *date* type and conventionally each period is identified by its last day:

DS_2		
Id_1	**Id_2**	**Me_1**
A	2010-12-31	"hello world"
A	2012-12-31	"say hello"
A	2013-12-31	"he"
B	2011-12-31	"hi, hello! "
B	2012-12-31	"hi”
B	2014-12-31	“hello!”

Example 3: DS_r := fill_time_series (DS_2, single) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010-12-31	"hello world"
A	2011-12-31	NULL
A	2012-12-31	"say hello"
A	2013-12-31	"he"
B	2011-12-31	"hi, hello! "

B	2012-12-31	"hi"
B	2013-12-31	NULL
B	2014-12-31	"hello!"

Example 4: DS_r := fill_time_series (DS_2, all) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010-12-31	"hello world"
A	2011-12-31	NULL
A	2012-12-31	"say hello"
A	2013-12-31	"he"
A	2014-12-31	NULL
B	2010-12-31	NULL
B	2011-12-31	"hi, hello! "
B	2012-12-31	"hi"
B	2013-12-31	NULL
B	2014-12-31	"hello!"

Given the Data Set DS_3, which contains *yearly* time series, where Id_2 is the reference time Identifier of *time_period* type:

DS_3		
Id_1	**Id_2**	**Me_1**
A	2010Y	"hello world"
A	2012Y	"say hello"
A	2013Y	"he"
B	2011Y	"hi, hello! "
B	2012Y	"hi"
B	2014Y	"hello!"

Example 5: DS_r := fill_time_series (DS_3, single) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010Y	"hello world"
A	2011Y	NULL
A	2012Y	"say hello"
A	2013Y	"he"
B	2011Y	"hi, hello! "
B	2012Y	"hi"
B	2013Y	NULL
B	2014Y	"hello!"

Example 6: DS_r := fill_time_series (DS_3, all) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010Y	"hello world"
A	2011Y	NULL
A	2012Y	"say hello"
A	2013Y	"he"
A	2014Y	NULL
B	2010Y	NULL
B	2011Y	"hi, hello! "
B	2012Y	"hi”
B	2013Y	NULL
B	2014Y	“hello!”

Given the Data Set DS_4, which contains both *quarterly* and *annual* time series relevant to the same phenomenon “A”, where Id_2 is the reference time Identifier of *time_period* type,:

DS_4		
Id_1	**Id_2**	**Me_1**
A	2010Y	"hello world"
A	2012Y	"say hello"
A	2010Q1	“he”
A	2010Q2	"hi, hello! "
A	2010Q4	"hi”
A	2011Q2	“hello!”

Example 7: DS_r := fill_time_series (DS_4, single) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010Y	"hello world"
A	2011Y	NULL
A	2012Y	"say hello"
A	2010Q1	“he”
A	2010Q2	"hi, hello! "
A	2010Q3	NULL
A	2010Q4	"hi”
A	2011Q2	“hello!”

Example 8: DS_r := fill_time_series (DS_4, all) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010Y	"hello world"
A	2011Y	NULL
A	2012Y	"say hello"
A	2010Q1	“he”
A	2010Q2	"hi, hello! "
A	2010Q3	NULL
A	2010Q4	"hi”
A	2011Q1	NULL
A	2011Q2	“hello!”
A	2011Q3	NULL
A	2011Q4	NULL
A	2012Q1	NULL
A	2012Q2	NULL
A	2012Q3	NULL
A	2012Q4	NULL

Flow to stock : flow_to_stock

Syntax

flow_to_stock (** op **)

Input Parameters

op the operand

Examples of valid syntaxes

flow_to_stock (ds_1)

Semantics for scalar operations

This operator does not perform scalar operations.

Input parameters type:

op :: dataset { identifier < time > _ , identifier _* , measure<number> _+ }

Result type:

result :: dataset { identifier < time > _ , identifier _* , measure<number> _+ }

Additional constraints

The operand dataset has an Identifier of type *time, date* or *time_period* and may have other Identifiers.

Behaviour

The statistical data that describe the "state" of a phenomenon on a given moment (e.g. resident population on a given moment) are often referred to as "stock data".
On the contrary, the statistical data that describe "events" which can happen continuously (e.g. changes in the resident population, such as births, deaths, immigration, emigration), are often referred to as "flow data".
This operator takes in input a Data Set which are interpreted as flows and calculates the change of the corresponding stock since the beginning of each time series by summing the relevant flows. In other words, the operator perform the cumulative sum from the first Data Point of each time series to each other following Data Point of the same time series.
The flow_to_stock operator can be applied only on Data Sets of time series and returns a Data Set of time series.
The result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set and contains the same time series as the operand, because the time series Identifiers (all the Identifiers except the reference time Identifier) are not changed.
As mentioned in the section "Behaviour of the Time Operators", the operator is assumed to know which is the *time* Identifier as well as the *period* of each time series.

Examples

Given the Data Set DS_1, which contains *yearly* time series, where Id_2 is the reference time Identifier of *time* type:

DS_1		
Id_1	**Id_2**	**Me_1**
A	2010-01/2010-12	2
A	2011-01/2011-12	5
A	2012-01/2012-12	-3
A	2013-01/2013-12	9
B	2010-01/2010-12	4

B	2011-01/2011-12	-8
B	2012-01/2012-12	0
B	2013-01/2013-12	6

Example 1: DS_r := flow_to_stock (DS_1) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010-01/2010-12	2
A	2011-01/2011-12	7
A	2012-01/2012-12	4
A	2013-01/2013-12	13
B	2010-01/2010-12	4
B	2011-01/2011-12	-4
B	2012-01/2012-12	-4
B	2013-01/2013-12	2

Given the Data Set DS_2, which contains *yearly* time series, where Id_2 is the reference time Identifier of *date* type (conventionally each period is identified by its last day):

DS_2		
Id_1	**Id_2**	**Me_1**
A	2010-12-31	2
A	2011-12-31	5
A	2012-12-31	-3
A	2013-12-31	9
B	2010-12-31	4
B	2011-12-31	-8
B	2012-12-31	0
B	2013-12-31	6

Example 2: DS_r := flow_to_stock (DS_2) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010-12-31	2
A	2011-12-31	7
A	2012-12-31	4
A	2013-12-31	13
B	2010-12-31	4
B	2011-12-31	-4
B	2012-12-31	-4
B	2013-12-31	2

Given the Data Set DS_3, which contains *yearly* time series, where Id_2 is the reference time Identifier of *time_period* type:

DS_3		
Id_1	**Id_2**	**Me_1**
A	2010Y	2
A	2011Y	5
A	2012Y	-3
A	2013Y	9
B	2010Y	4
B	2011Y	-8
B	2012Y	0
B	2013Y	6

Example 3: DS_r := flow_to_stock (DS_3) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010Y	2
A	2011Y	7
A	2012Y	4
A	2013Y	13
B	2010Y	4
B	2011Y	-4
B	2012Y	-4
B	2013Y	2

Given the Data Set DS_4, which contains both *quarterly* and *annual* time series relevant to the same phenomenon "A", where Id_2 is the reference time Identifier of *time_period* type:

DS_4		
Id_1	**Id_2**	**Me_1**
A	2010Y	2
A	2011Y	7
A	2012Y	4
A	2013Y	13
A	2010Q1	2
A	2010Q2	-3
A	2010Q3	7
A	2010Q4	-4

Example 4: DS_r := flow_to_stock (DS_3) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010Y	2
A	2011Y	9
A	2012Y	13
A	2013Y	26
A	2010Q1	2
A	2010Q2	-1
A	2010Q3	6
A	2010Q4	2

Stock to flow : **stock_to_flow**

Syntax

stock_to_flow (op)

Input parameters

op the operand

Examples of valid syntaxes

stock_to_flow (ds_1)

Semantics for scalar operations

This operator does not perform scalar operations.

Input parameters type:

op :: dataset { identifier < time > _ , identifier _* , measure<number> _+ }

Result type:

result :: dataset { identifier < time > _ , identifier _* , measure<number> _+ }

Additional constraints

The operand dataset has an Identifier of type *time, date* or *time_period* and may have other Identifiers.

Behaviour

The statistical data that describe the "state" of a phenomenon on a given moment (e.g. resident population on a given moment) are often referred to as "stock data".
On the contrary, the statistical data that describe "events" which can happen continuously (e.g. changes in the resident population, such as births, deaths, immigration, emigration), are often referred to as "flow data".
This operator takes in input a Data Set of time series which is interpreted as stock data and, for each time series, calculates the corresponding flow data by subtracting from the measure values of each regular period the corresponding measure values of the previous one.
The stock_to_flow operator can be applied only on Data Sets of time series and returns a Data Set of time series.
The result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set and contains the same time series as the operand, because the time series Identifiers (all the Identifiers except the reference time Identifier) are not changed.
The Attribute propagation rule is not applied.
As mentioned in the section "Behaviour of the Time Operators", the operator is assumed to know which is the *time* Identifier as well as the *period* of each time series.

Examples

Given the Data Set DS_1, which contains *yearly* time series, where Id_2 is the reference time Identifier of *time* type:

DS_1		
Id_1	**Id_2**	**Me_1**
A	2010-01/2010-12	2
A	2011-01/2011-12	7
A	2012-01/2012-12	4
A	2013-01/2013-12	13
B	2010-01/2010-12	4
B	2011-01/2011-12	-4
B	2012-01/2012-12	-4
B	2013-01/2013-12	2

Example 1: DS_r := stock_to_flow (DS_1) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010-01/2010-12	2
A	2011-01/2011-12	5
A	2012-01/2012-12	-3
A	2013-01/2013-12	9
B	2010-01/2010-12	4
B	2011-01/2011-12	-8
B	2012-01/2012-12	0
B	2013-01/2013-12	6

Given the Data Set DS_2, which contains *yearly* time series, where Id_2 is the reference time Identifier of *date* type (conventionally each period is identified by its last day):

DS_2		
Id_1	**Id_2**	**Me_1**
A	2010-12-31	2
A	2011-12-31	7
A	2012-12-31	4
A	2013-12-31	13
B	2010-12-31	4
B	2011-12-31	-4
B	2012-12-31	-4
B	2013-12-31	2

Example 2: DS_r := stock_to_flow (DS_2) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010-12-31	2
A	2011-12-31	5
A	2012-12-31	-3
A	2013-12-31	9
B	2010-12-31	4
B	2011-12-31	-8
B	2012-12-31	0
B	2013-12-31	6

Given the Data Set DS_3, which contains *yearly* time series, where Id_2 is the reference time Identifier of *time_period* type:

DS_3		
Id_1	**Id_2**	**Me_1**
A	2010Y	2
A	2011Y	7
A	2012Y	4
A	2013Y	13
B	2010Y	4
B	2011Y	-4
B	2012Y	-4
B	2013Y	2

Example 3: DS_r := stock_to_flow (DS_3) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010Y	2
A	2011Y	5
A	2012Y	-3
A	2013Y	9
B	2010Y	4
B	2011Y	-8
B	2012Y	0
B	2013Y	6

Given the Data Set DS_4, which contains both *quarterly* and *annual* time series relevant to the same phenomenon "A", where Id_2 is the *time* Identifier of *time_period* type:

DS_4		
Id_1	**Id_2**	**Me_1**
A	2010Y	2
A	2011Y	9
A	2012Y	13
A	2013Y	26
A	2010Q1	2
A	2010Q2	-1
A	2010Q3	6
A	2010Q4	2

Example 4: DS_r := stock_to_flow (DS_4) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2010Y	2
A	2011Y	7
A	2012Y	4
A	2013Y	13
A	2010Q1	2
A	2010Q2	-3
A	2010Q3	7
A	2010Q4	-4

Time shift : **timeshift**

Syntax

timeshift (op , shiftNumber)

Input parameters

op the operand
shiftNumber the number of periods to be shifted

Examples of valid syntaxes

timeshift (DS_1, 2)
timeshift (DS_1)

Semantics for scalar operations

This operator does not perform scalar operations.

Input parameters type:

op :: dataset { identifier < time > _ , identifier _* }
shiftNumber :: integer

Result type:

result :: dataset { identifier < time > _ , identifier _* }

Additional constraints

The operand dataset has an Identifier of type *time*, *date* or *time_period* and may have other Identifiers.

.

Behaviour

This operator takes in input a Data Set of time series and, for each time series of the Data Set, shifts the reference time Identifier of a number of periods (of the time series) equal to the shift_number parameter. If shift_number is negative, the shift is in the past, otherwise in the future. For example, if the period of the time series is month and shift_number is -1 the reference time Identifier is shifted of two months in the past.
The operator can be applied only on Data Sets of time series and returns a Data Set of time series.
The result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set and contains the same time series as the operand, because the time series Identifiers (all the Identifiers except the reference time Identifier) are not changed.
The Attribute propagation rule is not applied.
As mentioned in the section "Behaviour of the Time Operators", the operator is assumed to know which is the *time* Identifier as well as the *period* of each data point.

Examples

Given the Data Set DS_1, which contains *yearly* time series, where Id_2 is the reference time Identifier of *time* type:

DS_1		
Id_1	**Id_2**	**Me_1**
A	2010-01/2010-12	"hello world"
A	2011-01/2011-12	NULL
A	2012-01/2012-12	"say hello"
A	2013-01/2013-12	"he"
B	2010-01/2010-12	"hi, hello! "
B	2011-01/2011-12	"hi"
B	2012-01/2012-12	NULL
B	2013-01/2013-12	"hello!"

Example 1: DS_r := time_shift (DS_1 , -1) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2009-01/2009-12	"hello world"
A	2010-01/2010-12	NULL
A	2011-01/2011-12	"say hello"
A	2012-01/2012-12	"he"
B	2009-01/2009-12	"hi, hello! "
B	2010-01/2010-12	"hi"
B	2011-01/2011-12	NULL
B	2012-01/2012-12	"hello!"

Given the Data Set DS_2, which contains *yearly* time series, where Id_2 is the reference time Identifier of *date* type (conventionally each period is identified by its last day):

DS_2		
Id_1	**Id_2**	**Me_1**
A	2010-12-31	"hello world"
A	2011-12-31	NULL
A	2012-12-31	"say hello"
A	2013-12-31	"he"
B	2010-12-31	"hi, hello! "
B	2011-12-31	"hi”
B	2012-12-31	NULL
B	2013-12-31	“hello!”

Example 2: DS_r := time_shift (DS_2 , 2) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2012-12-31	"hello world"
A	2013-12-31	NULL
A	2014-12-31	"say hello"
A	2015-12-31	"he"
B	2012-12-31	"hi, hello! "
B	2013-12-31	"hi”
B	2014-12-31	NULL
B	2015-12-31	“hello!”

Given the Data Set DS_3, which contains *yearly* time series, where Id_2 is the reference time Identifier of *time_period* type:

DS_3		
Id_1	**Id_2**	**Me_1**
A	2010Y	"hello world"
A	2011Y	NULL
A	2012Y	"say hello"
A	2013Y	"he"
B	2010Y	"hi, hello! "
B	2011Y	"hi”
B	2012Y	NULL
B	2013Y	“hello!”

Example 3: DS_r := time_shift (DS_3 , 1) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2011Y	"hello world"

A	2012Y	NULL
A	2013Y	"say hello"
A	2014Y	"he"
B	2011Y	"hi, hello! "
B	2012Y	"hi”
B	2013Y	NULL
B	2014Y	“hello!”

Given the Data Set DS_4, which contains both *quarterly* and *annual* time series relevant to the same phenomenon “A”, where Id_2 is the reference time Identifier of *time_period* type:

DS_4		
Id_1	**Id_2**	**Me_1**
A	2010Y	"hello world"
A	2011Y	NULL
A	2012Y	"say hello"
A	2013Y	"he"
A	2010Q1	"hi, hello! "
A	2010Q2	"hi”
A	2010Q3	NULL
A	2010Q4	“hello!”

Example 4: DS_r := time_shift (DS_3 , -1) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
A	2009Y	"hello world"
A	2010Y	NULL
A	2011Y	"say hello"
A	2012Y	"he"
A	2009Q4	"hi, hello! "
A	2010Q1	"hi”
A	2010Q2	NULL
A	2010Q3	“hello!”

Time aggregation : **time_agg**

The operator **time_agg** converts *time, date* and *time_period* values from a smaller to a larger duration.

Syntax

time_agg (periodIndTo { **,** periodIndFrom } { **,** op } { **, first | last** } **)**

Input parameters

op	the scalar value, the Component or the Data Set to be converted. If not specified, then **time_agg** is used in combination within an aggregation operator
periodIndFrom	the source period indicator
periodIndTo	the target period indicator

Examples of valid syntaxes

```
sum ( DS group all time_agg ( Me, “A” ) )
time_agg ( “A”, cast ( “2012Q1”, time_period , ”YYYY\Qq” ) )
time_agg(“M”, cast (“2012-12-23”, date, “YYYY-MM-DD”) )
time_agg(“M”, DS1)
ds_2 := ds1[calc Me1 := time_agg(“M”,Me1)]
```

Semantics for scalar operations

The operator converts a *time, date* or *time_period* value from a smaller to a larger duration.

Input parameters type

op :: dataset { identifier < time > _ , identifier _* }
| component<time>
| time
periodIndFrom :: duration
periodIndTo :: duration

Result type

op :: dataset { identifier < time > _ , identifier _* }
| component<time>
| time

Additional constraints

If op is a Data Set then it has exactly an Identifier of type *time, date* or *time_period* and may have other Identifiers. It is only possible to convert smaller duration values to larger duration values (e.g. it is possible to convert *monthly* data to *annual* data but the contrary is not allowed).

Behaviour

The scalar version of this operator takes as input a *time*, *date* or *time_period* value, converts it to periodIndTo and returns a scalar of the corresponding type.
The Data Set version acts on a single Measure Data Set of type *time*, *date* or *time_per*iod and returns a Data Set having the same structure.
Finally, VTL also provides a component version, for use in combination with an aggregation operator, because the change of frequency requires an aggregation. In this case, the operator converts the **period_indicator** of the data points (e.g., convert *monthly* data to *annual* data).
On *time* type, the operator maps the input value into the comprising larger regular interval, whose duration is the one specified by the periodIndTo parameter.
On *date* type, the operator maps the input value into the comprising larger period, whose duration is the one specified by the periodIndTo parameter, which is conventionally represented either by the start or by the end date, according to the **first/last** parameter.
On *time_period* type, the operator maps the input value into the comprising larger time period specified by the periodIndTo parameter (the original period indicator is converted in the target one and the number of periods is adjusted correspondingly).
The input duration periodIndFrom is optional. In case of *time_period* Data Points, the input duration can be inferred from the internal representation of the value. In case of *time* or *date* types, it is inferred by the implementation. Filters on input time series can be obtained with the **filter** clause.

Examples

Given the Data Set DS_1

DS_1		
Id_1	**Id_2**	**Me_1**
2010Q1	A	20
2010Q2	A	20
2010Q3	A	20
2010Q1	B	50
2010Q2	B	50
2010Q1	C	10
2010Q2	C	10

Example 1: DS_r := sum (DS_1) group all time_agg (“A” , _ , Me_1) results in:

DS_r		
Id_1	**Id_2**	**Me_1**
2010	A	60
2011	B	100
2010	C	20

Example 2: DS_r := time _agg (“Q”, cast (“2012M01”, time_period, ”YYYY\MMM”))

Returns: “2012Q1”.

Example 3: The following example maps a *date* to quarter level, 2012 (end of the period).

time_agg(“Q”, cast(“20120213”, date, ”YYYYMMDD”), _ , false)

and produces a date value corresponding to the *string* “20120331”

Example 4: The following example maps a *date* to year level, 2012 (beginning of the period).

time_agg(cast(”A”, “2012M1”, date, ”YYYYMMDD”), _, true)

and produces a *date* value corresponding to the string *“20120101”.*

Actual time : **current_date**

Syntax

current_date ( )

Input parameters

None

Examples of valid syntax

current_date

Semantics for scalar operations

The operator **current_date** returns the current time as a *date* type.

Input parameters type

This operator has no input parameters.

Result type

result :: date

Additional constraints

None.

Behaviour

The operator return the current date

Examples

cast (current_date, string, "YYYY.MM.DD")

VTL-ML - Set operators

Union: union

Syntax

union (dsList **)**

dsList ::= ds { **,** ds }*

Input parameters

dsList the list of Data Sets in the union

Examples of valid syntaxes

union (ds2, ds3)

Semantics for scalar operations

This operator does not perform scalar operations.

Input parameters type

ds :: dataset

Result type

result :: dataset

Additional constraints

All the Data Sets in dsList have the same Identifier, Measure and Attribute Components.

Behaviour

The **union** operator implements the union of functions (i.e., Data Sets). The resulting Data Set has the same Identifier, Measure and Attribute Components of the operand Data Sets specified in the dsList, and contains the Data Points belonging to any of the operand Data Sets.

The operand Data Sets can contain Data Points having the same values of the Identifiers. To avoid duplications of Data Points in the resulting Data Set, those Data Points are filtered by chosing the Data Point belonging to the left most operand Data Set. For instance, let's assume that in **union** (ds1, ds2) the operand ds1 contains a Data Point dp1 and the operand ds2 contains a Data Point dp2 such that dp1 has the same Identifiers values of dp2, then the resulting Data Set contains dp1 only.

The operator has the typical behaviour of the "Behaviour of the Set operators" (see the section "Typical behaviours of the ML Operators").

The automatic Attribute propagation is not applied.

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	5
2012	G	Total	Total	2
2012	F	Total	Total	3

DS_2				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**

2012	N	Total	Total	23
2012	S	Total	Total	5

Example 1: DS_r := union(DS_1,DS_2) results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	5
2012	G	Total	Total	2
2012	F	Total	Total	3
2012	N	Total	Total	23
2012	S	Total	Total	5

Given the operand Data Sets DS_1 and DS_2:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	5
2012	G	Total	Total	2
2012	F	Total	Total	3

DS_2				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	23
2012	S	Total	Total	5

Example 2: DS_r := union (DS_1, DS_2) results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	5
2012	G	Total	Total	2
2012	F	Total	Total	3
2012	S	Total	Total	5

Intersection : intersect

Syntax

intersect (dsList **)**

dsList ::= ds { **,** ds }*

Input parameters

dsList the list of Data Sets in the intersection

Examples of valid syntaxes

intersect (ds2, ds3)

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

ds :: dataset

Return type

result :: dataset

Additional constraints

All the Data Sets in dsList have the same Identifier, Measure and Attribute Components.

Behaviour

The **intersect** operator implements the intersection of functions (i.e., Data Sets). The resulting Data Set has the same Identifier, Measure and Attribute Components of the operand Data Sets specified in the dsList, and contains the Data Points belonging to all the operand Data Sets.
The operand Data Sets can contain Data Points having the same values of the Identifiers. To avoid duplications of Data Points in the resulting Data Set, those Data Points are filtered by chosing the Data Point belonging to the left most operand Data Set. For instance, let's assume that in **intersect** (ds1, ds2) the operand ds1 contains a Data Point dp1 and the operand ds2 contains a Data Point dp2 such that dp1 has the same Identifiers values of dp2, then the resulting Data Set contains dp1 only.
The operator has the typical behaviour of the "Behaviour of the Set operators" (see the section "Typical behaviours of the ML Operators").
The automatic Attribute propagation is not applied.

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	1
2012	G	Total	Total	2
2012	F	Total	Total	3

DS_2				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2011	B	Total	Total	10
2012	G	Total	Total	2
2011	M	Total	Total	40

Example 1: DS_r := intersect(DS_1,DS_2) results in:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	G	Total	Total	2

Set difference : setdiff

Syntax

setdiff (ds1, ds2 **)**

Input parameters

ds1 the first Data Set in the difference (the minuend)
ds2 the second Data Set in the difference (the subtrahend)

Examples of valid syntaxes

setdiff (ds2, ds3)

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

ds1, ds2 :: dataset

Result type

result :: dataset

Additional constraints

The operand Data Sets have the same Identifier, Measure and Attribute Components.

Behaviour

The operator implements the set difference of functions (i.e. Data Sets), interpreting the Data Points of the input Data Sets as the elements belonging to the operand sets, the minuend and the subtrahend, respectively. The operator returns one single Data Set, with the same Identifier, Measure and Attribute Components as the operand Data Sets, containing the Data Points that appear in the first Data Set but not in the second. In other words, for setdiff (ds1, ds2), the resulting Dataset contains all the data points Data Point dp1 of the operand ds1 such that there is no Data Point dp2 of ds2 having the same values for homonym Identifier Components.
The operator has the typical behaviour of the "Behaviour of the Set operators" (see the section "Typical behaviours of the ML Operators").
The automatic Attribute propagation is not applied.

Examples

Given the operand Data Sets DS_1 and DS_2:

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	10
2012	G	Total	Total	20
2012	F	Total	Total	30
2012	M	Total	Total	40
2012	I	Total	Total	50
2012	S	Total	Total	60

DS_2				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2011	B	Total	Total	10
2012	G	Total	Total	20

2012	F	Total	Total	30
2012	M	Total	Total	40
2012	I	Total	Total	50
2012	S	Total	Total	60

Example 1: DS_r := setdiff (DS_1, DS_2) results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	10

Given the operand Data Sets DS_1 and DS_2 :

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
R	M	2011	7
R	F	2011	10
R	T	2011	12

DS_2			
Id_1	**Id_2**	**Id_3**	**Me_1**
R	M	2011	7
R	F	2011	10

Example 2: DS_r := setdiff (DS_1 , DS_2) results in:

DS_r			
Id_1	**Id_2**	**Id_3**	**Me_1**
R	T	2011	12

Simmetric difference : **symdiff**

Syntax

symdiff (ds1, ds2 **)**

Input parameters

ds1 the first Data Set in the difference

ds2 the second Data Set in the difference

Examples of valid syntaxes

symdiff (ds_2, ds_3)

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

ds1, ds2 :: dataset

Result type

result :: dataset

Additional constraints

The operand Data Sets have the same Identifier, Measure and Attribute Components.

Behaviour

The operator implements the symmetric set difference between functions (i.e. Data Sets), interpreting the Data Points of the input Data Sets as the elements in the operand Sets. The operator returns one Data Set, with the same Identifier, Measure and Attribute Components as the operand Data Sets, containing the Data Points that appear in the first Data Set but not in the second and the Data Points that appear in the second Data Set but not in the first one.

Data Points are compared to one another by Identifier Components. For symdiff (ds1, ds2), the resulting Data Set contains all the Data Points dp1 contained in ds1 for which there is no Data Point dp2 in ds2 with the same values for homonym Identifier components and all the Data Points dp2 contained in ds2 for which there is no Data Point dp1 in ds1 with the same values for homonym Identifier Components.

The operator has the typical behaviour of the "Behaviour of the Set operators" (see the section "Typical behaviours of the ML Operators").

The automatic Attribute propagation is not applied.

Examples

Given the operand Data Sets DS_1 and DS_2 :

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	1
2012	G	Total	Total	2
2012	F	Total	Total	3
2012	M	Total	Total	4
2012	I	Total	Total	5
2012	S	Total	Total	6

DS_2				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2011	B	Total	Total	1
2012	G	Total	Total	2
2012	F	Total	Total	3
2012	M	Total	Total	4
2012	I	Total	Total	5
2012	S	Total	Total	6

Example 1: DS_r := symdiff (DS_1, DS_2) results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	1
2011	B	Total	Total	1

VTL-ML - Hierarchical aggregation

Hierarchical roll-up : **hierarchy**

Syntax

hierarchy (op , hr { **condition** condComp { **,** condComp }* } { **rule** ruleComp } { mode } { input } { output } **)**

mode ::= **non_null | non_zero | partial_null | partial_zero | always_null | always_zero**

input ::= **dataset | rule | rule_priority**

output ::= **computed | all**

Input parameters

op	the operand Data Set.
hr	the hierarchical Ruleset to be applied.
condComp	condComp is a Component of op to be associated (in positional order) to the conditioning Value Domains or Variables defined in hr (if any).
ruleComp	ruleComp is the Identifier of op to be associated to the rule Value Domain or Variable defined in hr.
mode	this parameter specifies how to treat the possible missing Data Points corresponding to the Code Items in the right side of a rule and which Data Points are produced in output. The meaning of the possible values of the parameter is explained below.
input	this parameter specifies the source of the values used as input of the hierarchical rules. The meaning of the possible values of the parameter is explained below.
output	this parameter specifies the content of the resulting Data Set. The meaning of the possible values of the parameter is explained below.

Examples of valid syntaxes

hierarchy (DS1, HR1 rule Id_1 non_null all)
hierarchy (DS2, HR2 condition Comp_1, Comp_2 rule Id_3 non_zero rule computed)

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

op ::	dataset { measure<number> _ }
hr ::	name < hierarchical >
condComp ::	name < component >
ruleComp ::	name < dentifier >

Result type

result ::	dataset {measure<number> _ }

Additional constraints

If hr is defined on Value Domains then it is mandatory to specify the condition (if any) and the rule parameters. Moreover, the Components specified as condComp and ruleComp must belong to the operand op and must take values on the Value Domains corresponding, in positional order, to the ones specified in the condition and rule parameter of hr.

If hr is defined on Variables, the specification of condComp and ruleComp is not needed, but they can be specified all the same if it is desired to show explicitly in the invocation which are the involved Components: in this case, the condComp and ruleComp must be the same and in the same order as the Variables specified in in the condition and rule signatures of hr.

Behaviour

The **hierarchy** operator applies the rules of hr to op as specified in the parameters. The operator returns a Data Set with the same Identifiers and the same Measure as op. The Attribute propagation rule is applied on the groups of Data Points which contribute to the same Data Points of the result.

The behaviours relevanto to the different options of the input parameters are the following.

First, the parameter **input** is considered to determine the source of the Data Points used as input of the Hierarchy. The possible options of the parameter input and the corresponding behaviours are the following:

dataset — For each Rule of the Ruleset and for each item on the right hand side of the Rule, the operator takes the input Data Points exclusively from the operand op.

rule — For each Rule of the Ruleset and for each item on the right-hand side of the Rule:

- if the item is not defined as the result (left-hand side) of another Rule, the current Rule takes the input Data Points from the operand op
- if the item is defined as the result of another Rule, the current Rule takes the input Data Points from the computed output of such other Rule;

rule_priority — For each Rule of the Ruleset and for each item on the right-hand side of the Rule:

- if the item is not defined as the result (left-hand side) of another rule, the current Rule takes the input Data Points from the operand op.
- if the item is defined as the result of another Rule, then:
    - if an expected input Data Point exists in the computed output of such other Rule and its Measure is not NULL, then the current Rule takes such Data Point;
    - if an expected input Data Point does not exist in the computed output of such other Rule or its measure is NULL, then the current Rule takes the Data Point from op (if any) having the same values of the Identifiers;

if the parameter input is not specified then it is assumed to be rule.

Then the parameter mode is considered, to determine the behaviour for missing Data Points and for the Data Points to be produced in the output. The possible options of the parameter mode and the corresponding behaviours are the following:

non_null — the result Data Point is produced when its computed Measure value is not NULL (i.e., when no Data Point corresponding to the Code Items of the right side of the rule is missing or has NULL Measure value); in the calculation, the possible missing Data Points corresponding to the Code Items of the right side of the rule are considered existing and having a Measure value equal to NULL;

non_zero — the result Data Point is produced when its computed Measure value is not equal to 0 (zero); the possible missing Data Points corresponding to the Code Items of the right side of the rule are considered existing and having a Measure value equal to 0;

partial_null — the result Data Point is produced if at least one Data Point corresponding to the Code Items of the right side of the rule is found (whichever is its Measure value); the possible missing Data Points corresponding to the Code Items of the right side of the rule are considered existing and having a NULL Measure value;

partial_zero — the result Data Point is produced if at least one Data Point corresponding to the Code Items of the right side of the rule is found (whichever is its Measure value); the possible missing Data Points corresponding to the Code Items of the right side of the rule are considered existing and having a Measure value equal to 0 (zero);

always_null — the result Data Point is produced in any case; the possible missing Data Points corresponding to the Code Items of the right side of the rule are considered existing and having have a Measure value equal to NULL;

always_zero — the result Data Point is produced in any case; the possible missing Data Points corresponding to the Code Items of the right side of the rule are considered existing and having a Measure value equal to 0 (zero);

If the parameter mode is not specified, then it is assumed to be non_null

The following table summarizes the behaviour of the options of the parameter "mode"

OPTION of the MODE PARAMETER:	Missing Data Points are considered:	Null Data Points are considered:	Condition for evaluating the rule	Returned Data Points
Non_null	NULL	NULL	If all the involved Data Points are not NULL	Only not NULL Data Points (Zeros are returned too)
Non_zero	Zero	NULL	If at least one of the involved Data Points is <> zero	Only not zero Data Points (NULLS are returned too)

Partial_null	NULL	NULL	If at least one of the involved Data Points is not NULL	Data Points of any value (NULL, not NULL and zero too)
Partial_zero	Zero	NULL	If at least one of the involved Data Points is not NULL	Data Points of any value (NULL, not NULL and zero too)
Always_null	NULL	NULL	Always	Data Points of any value (NULL, not NULL and zero too)
Always_zero	Zero	NULL	Always	Data Points of any value (NULL, not NULL and zero too)

Finally the parameter output is considered, to determine the content of the resulting Data Set. The possible options of the parameter output and the corresponding behaviours are the following:

computed — the resulting Data Set contains only the set of Data Points computed according to the Ruleset

all — the resulting Data Set contains the union between the set of Data Points "R" computed according to the Ruleset and the set of Data Points of op that have different combinations of values for the Identifiers. In other words, the result is the outcome of the following (virtual) expression: union (setdiff (op , R) , R)

If the parameter output is not specified then it is assumed to be computed.

Examples

Given the following hierarchical ruleset:

```
define hierarchical ruleset  HR_1 ( valuedomain rule  VD_1 )  is
        A = J + K + L
      ; B = M + N + O
      ; C = P + Q
      ; D = R + S
      ; E = T + U + V
      ; F = Y + W + Z
      ; G = B + C
      ; H = D + E
      ; I  = D + G
end hierarchical ruleset
```

And given the operand Data Set DS_1 (where At_1 is viral and the propagation rule says that the alphabetic order prevails the NULL prevails on the alphabetic characters and the Attribute value for missing Data Points is assumed as NULL):

DS_1			
Id_1	**Id_2**	**Me_1**	**At_1**
2010	M	2	Dx
2010	N	5	Pz
2010	O	4	Pz
2010	P	7	Pz
2010	Q	-7	Pz
2010	S	3	Ay
2010	T	9	Bq
2010	U	NULL	Nj

2010	V	6	Ko

Example 1: DS_r := hierarchy (DS_1, HR_1 rule Id_2 non_null) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**At_1**
2010	B	11	Dx
2010	C	0	Pz
2010	G	19	Dx

Example 2: DS_r := hierarchy (DS_1, HR_1 rule Id_2 non_zero) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**At_1**
2010	B	11	Dx
2010	D	3	NULL
2010	E	NULL	Bq
2010	G	11	Dx
2010	H	NULL	NULL
2010	I	14	NULL

Example 2: DS_r := hierarchy (DS_1, HR_1 rule Id_2 partial_null) results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**At_1**
2010	B	11	Dx
2010	C	0	Pz
2010	D	NULL	NULL
2010	E	NULL	Bq
2010	G	11	Dx
2010	H	NULL	NULL
2010	I	NULL	NULL

VTL-ML - Aggregate and Analytic operators

The following table lists the operators that can be invoked in the Aggregate or in the Analytic invocations described below and their main characteristics.

Operator	Description	Allowed invocations	Type of the resulting Measure	Type of the operand Measures
count	number of Data Points	Aggregate Analytic	integer	any
min	minimum value of a set of values	Aggregate Analytic	any	any
max	maximum value of a set of values	Aggregate Analytic	any	any
median	median value of a set of numbers	Aggregate Analytic	number	number
sum	sum of a set of numbers	Aggregate Analytic	number	number
avg	average value of a set of numbers	Aggregate Analytic	number	number
stddev_pop	population standard deviation of a set of numbers	Aggregate Analytic	number	number
stddev_samp	sample standard deviation of a set of numbers	Aggregate Analytic	number	number
var_pop	population variance of a set of numbers	Aggregate Analytic	number	number
var_samp	sample variance of a set of numbers	Aggregate Analytic	number	number
first_value	first value in an ordered set of values	Analytic	any	any
last_value	last value in an ordered set of values	Analytic	any	any
lag	in an ordered set of Data Points, it returns the value(s) taken from a Data Point at a given physical offset prior to the current Data Point	Analytic	any	any
lead	in an ordered set of Data Points, it returns the value(s) taken from a Data Point at a given physical offset beyond the current Data Point	Analytic	any	any
rank	rank (order number) of a Data Point in an ordered set of Data Points	Analytic	integer	any

ratio_to_report	ratio of a value to the sum of a set of values	Analytic	number	number

Aggregate invocation

Syntax

in a Data Set expression:

aggregateOperator **(** firstOperand { **,** additionalOperand }* { groupingClause } **)**

in a Component expression within an aggr clause)

aggregateOperator **(** firstOperand { **,** additionalOperand }* **)** { groupingClause }

aggregateOperator ::= **avg** | **count** | **max** | **median** | **min** | **stddev_pop**
| **stddev_samp** | **sum** | **var_pop** | **var_samp**

groupingClause ::= { **group by** groupingId {, groupingId}*
| **group except** groupingId {, groupingId}*
| **group all** conversionExpr }[1]
{ **having** havingCondition }

Input Parameters

aggregateOperator — the keyword of the aggregate operator to invoke (e.g., **avg**, **count**, **max** ...)

firstOperand — the first operand of the invoked aggregate operator (a Data Set for an invocation at Data Set level or a Component of the input Data Set for an invocation at Component level within a **aggr** operator or a **aggr** clause in a join operation)

additionalOperand — an additional operand (if any) of the invoked operator. The various operators can have a different number of parameters. The number of parameters, their types and if they are mandatory or optional depend on the invoked operator

groupingClause — the following alternative grouping options:

- **group by** — the Data Points are grouped by the values of the specified Identifiers (groupingId). The Identifiers not specified are dropped in the result.
- **group except** — the Data Points are grouped by the values of the Identifiers not specified as groupingId. The Identifiers specified as groupingId are dropped in the result.
- **group all** — converts the values of an Identifier Component using conversionExpr and keeps all the resulting Identifiers.

groupingId — Identifier Component to be kept (in the **group by** clause) or dropped (in the **group except** clause).

conversionExpr — specifies a conversion operator (e.g., **time_agg**) to convert data from finer to coarser granularity. The conversion operator is applied on an Identifier of the operand Data Set op.

havingCondition — a condition (*boolean* expression) at component level, having only Components of the input Data Sets as operands (and possibly constants), to be fulfilled by the groups of Data Points: only groups for which havingCondition evaluates to TRUE appear in the result. The havingCondition refers to the groups specified through the groupingClause, therefore it must invoke aggregate operators (e.g. **avg**, **count**, **max** ..., see also the corresponding sections). A correct example of havingCondition is:

max(obs_value) < 1000

while the condition obs_value < 1000 is not a right havingCondition, because it refers to the values of single Data Points and not to the groups. The count operator is used in a havingCondition without parameters, e.g.:

sum (ds group by id1 having count ( ) >= 10)

Examples of valid syntaxes

avg (DS_1)
avg (DS_1 group by Id_1, Id_2)

```
avg ( DS_1 group except  Id_1, Id_2  )
avg ( DS_1 group all time_agg ( "Q" ) )
```

Semantics for scalar operations

The aggregate operators cannot be applied to scalar values.

Input parameters type

firstOperand ::	dataset | component
additionalOperand ::	see the type of the additional parameter (if any) of the invoked aggregateOperator. The aggregate operators and their parameters are described in the following sections.
groupingId ::	name < identifier >
conversionExpr ::	identifier
havingCondition ::	component<boolean>

Result type:

result ::	dataset | component

Additional constraints

The Aggregate invocation cannot be nested in other Aggregate or Analytic invocations.
The aggregate operations at component level can be invoked within the **aggr** clause, both as part of a join operator and the **aggr** operator (see the parameter aggrExpr of those operators).
The basic scalar types of firstOperand and additionalOperand (if any) must be compliant with the specific basic scalar types required by the invoked operator (the required basic scalar types are described in the table at the beginning of this chapter and in the sections of the various operators below).
The conversionExpr parameter applies just one conversion operator to just one Identifier belonging to the input Data Set. The basic scalar type of the Identifier must be compatible with the basic scalar type of the conversion operator.
If the grouping clause is omitted, then all the input Data Points are aggregated in a single group and the clause returns a Data Set that contains a single Data Point and has no Identifiers.

Behaviour

The aggregateOperator is applied as usual to all the measures of the firstOperand Data Set (if invoked at Data Set level) or to the firstOperand Component of the input Data Set (if invoked at Component level). In both cases, the operator calculates the required aggregated values for groups of Data Points of the input Data Set. The groups of Data Points to be aggregated are specified through the groupingClause, which allows the following alternative options.

group by	the Data Points are grouped by the values of the specified Identifiers. The Identifiers not specified are dropped in the result.
group except	the Data Points are grouped by the values of the Identifiers not specified in the clause. The specified Identifiers are dropped in the result.
group all	converts an Identifier Component using conversionExpr and keeps all the Identifiers.

The **having** clause is used to filter groups in the result by means of an aggregate condition evaluated on the single groups (for example the minimum number of rows in the group).
If no grouping clause is specified, then all the input Data Points are aggregated in a single group and the operator returns a Data Set that contains a single Data Point and has no Identifiers.
For the invocation at Data Set level, the resulting Data Set has the same Measures as the operand. For the invocation at Component level, the resulting Data Set has the Measures explicitly calculated (all the other Measures are dropped because no aggregation behaviour is specified for them).
For invocation at Data Set level, the Attribute propagation rule is applied. For invocation at Component level, the Attributes calculated within the aggr clause are maintained in the result; for all the other Attributes that are defined as **viral**, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule section in the User Manual).
As mentioned, the Aggregate invocation at component level can be done within the **aggr** clause, both as part of a Join operator and the **aggr** operator (see the parameter aggrExpr of those operators), therefore, for a better comprehension fo the behaviour at Component level, see also those operators.

Examples

Given the Data Set DS_1

DS_1				
Id_1	**Id_2**	**Id_3**	**Me_1**	**At_1**
2010	E	XX	20	
2010	B	XX	1	H
2010	R	XX	1	A
2010	F	YY	23	
2011	E	XX	20	P
2011	B	ZZ	1	N
2011	R	YY	-1	P
2011	F	XX	20	Z
2012	L	ZZ	40	P
2012	E	YY	30	P

Example1: DS_r := avg (DS_1 group by Id_1) provided that At_1 is non viral, results in:

DS_r	
Id_1	**Me_1**
2010	11.25
2011	10
2012	35

Note: the example above can be rewritten equivalently in the following forms:

```
DS_r := avg ( DS_1 group except Id_2, Id_3 )
DS_r := avg ( DS_1#Me_1 group by Id_1 )
```

Example2: DS_r := sum (DS_1 group by Id_1, Id_3) provided that At_1 is non viral, results in:

DS_r		
Id_1	**Id_3**	**Me_1**
2010	XX	22
2010	YY	23
2011	XX	40
2011	ZZ	1
2011	YY	-1
2012	ZZ	40
2012	YY	30

Example3: DS_r := avg (DS_1) provided that At_1 is non viral results in:

DS_r
Me_1
15.5

Example4: DS_r := DS_1 [aggr Me_2 := max (Me_1) , Me_3 := min (Me_1) group by Id_1]

provided that At_1 is viral and the first letter in alphabetic order prevails and NULL prevails on all the other characters, results in:

DS_r			
Id_1	**Me_2**	**Me_3**	**At_1**
2010	23	1	
2011	20	-1	N
2012	40	30	P

Analytic invocation

Syntax

analyticOperator **(** firstOperand { **,** additionalOperand }* **over (** analyticClause **))**

analyticOperator ::= **avg** | **count** | **max** | **median** | **min** | **stddev_pop**
| **stddev_samp** | **sum** | **var_pop** | **var_samp**
| **first_value** | **lag** | **last_value** | **lead** | **rank** | **ratio_to_report**

analyticClause ::= { partitionClause } { orderClause } { windowClause }

partitionClause ::= **partition by** identifier { **,** identifier }*

orderClause ::= **order by** component { **asc** | **desc** } { **,** component { **asc** | **desc** } }*

windowClause ::= { **data points** | **range** }^{1} **between** limitClause **and** limitClause

limitClause ::= { num **preceding** | num **following** | **current data point** | **unbounded preceding** | **unbounded following** }^{1}

Parameters

analyticOperator — the keyword of the analytic operator to invoke (e.g., **avg**, **count**, **max** ...)

firstOperand — the first operand of the invoked analytic operator (a Data Set for an invocation at Data Set level or a Component of the input Data Set for an invocation at Component level within a **calc** operator or a **calc** clause in a join operation)

additionalOperand — an additional operand (if any) of the invoked operator. The various operators can have a different number of parameters. The number of parameters, their types and if they are mandatory or optional depend on the invoked operator

analyticClause — clause that specifies the analytic behaviour

partitionClause — clause that specifies how to partition Data Points in groups to be analysed separately. The input Data Set is partitioned according to the values of one or more Identifier Components. If the clause is omitted, then the Data Set is partitioned by the Identifier Components that are not specified in the orderClause.

orderClause — clause that specifies how to order the Data Points. The input Data Set is ordered according to the values of one or more Components, in ascending order if **asc** is specified, in descending order if **desc** is specified, by default in ascending order if the **asc** and **desc** keywords are omitted.

windowClause — clause that specifies how to apply a sliding window on the ordered Data Points. The keyword **data points** means that the sliding window includes a certain number of Data Points before and after the current Data Point in the order given by the orderClause. The keyword **range** means that the sliding windows includes all the Data Points whose values are in a certain range in respect to the value, for the current Data Point, of the Measure which the analytic is applied to.

limitClause — clause that can specify either the lower or the upper boundaries of the sliding window. Each boundary is specified in relationship either to the whole partition or to the current data point under analysis by using the following keywords:

- **unbounded preceding** means that the sliding window starts at the first Data Point of the partition (it make sense only as the first limit of the window)
- **unbounded following** indicates that the sliding window ends at the last Data Point of the partition (it makes sense only as the second limit of the window)
- **current data point** specifies that the window starts or ends at the current Data Point.
- num **preceding** specifies either the number of **data points** to consider preceding the current data point in the order given by the orderClause (when **data points** is specified in the window clause), or the maximum difference to consider, as for the Measure which the analytic is applied to, between the value of the current Data Point and the generic other Data Point (when **range** is specified in the windows clause).
- num **following** specifies either the number of data points to consider following the current data point in the order given by the orderClause (when **data points** is specified in the window clause), or the maximum difference to consider, as for the Measure which the analytic is applied to, between the values of the generic other Data Point and the current Data Point (when **range** is specified in the windows clause).

If the whole windowClause is omitted then the default is **data points between unbounded preceding and current data point**.

identifier — an Identifier Component of the input Data Set

component — a Component of the input Data Set

num — a scalar *number*

Examples of valid syntaxes

```
sum ( DS_1 over ( partition by  Id_1  order by  Id_2  ) )
sum ( DS_1 over ( order by  Id_2  ) )
avg ( DS_1 over ( order by  Id_1  data points between 1 preceding and 1 following ) )
DS_1 [ calc M1 := sum ( Me_1 over ( order by  Id_1  ) ) ]
```

Semantics for scalar operations

The analytic operators cannot be applied to scalar values.

Input parameters type

firstOperand ::	dataset
	| component
additionalOperand ::	see the type of the additional parameter (if any) of the invoked operator. The operators and their parameters are described in the following sections.
identifier ::	name < identifier >
component ::	name < component >
num ::	integer

Result type

result ::	dataset
	| component

Additional constraints

The analytic invocation cannot be nested in other Aggregate or Analytic invocations.
The analytic operations at component level can be invoked within the **calc** clause, both as part of a Join operator and the **calc** operator (see the parameter calcExpr of those operators).
The basic scalar types of firstOperand and additionalOperand (if any) must be compliant with the specific basic scalar types required by the invoked operator (the required basic scalar types are described in the table at the beginning of this chapter and in the sections of the various operators below).

Behaviour

The analytic Operator is applied as usual to all the Measures of the input Data Set (if invoked at Data Set level) or to the specified Component of the input Data Set (if invoked at Component level). In both cases, the operator calculates the desired output values for each Data Point of the input Data Set.

The behaviour of the analytic operations can be procedurally described as follows:

- The Data Points of the input Data Set are first partitioned (according to partitionBy) and then ordered (according to orderBy).
- The operation is performed for each Data Point (named “current Data Point”) of the input Data Set. For each input Data Point, one output Data Point is returned, having the same values of the Identifiers. The analytic operator is applied to a “window” which includes a set of Data Points of the input Data Set and returns the values of the Measure(s) of the output Data Point.
    - If windowClause is not specified, then the set of Data Points which contribute to the analytic operation is the whole partition which the current Data Point belongs to
    - If windowClause is specified, then the set of Data Points is the one specified by windowClause (see windowsClause and LimitClause explained above).

For the invocation at Data Set level, the resulting Data Set has the same Measures as the input Data Set firstOperand. For the invocation at Component level, the resulting Data Set has the Measures of the input Data Set plus the Measures explicitly calculated through the **calc** clause.

For the invocation at Data Set level, the Attribute propagation rule is applied. For invocation at Component level, the Attributes calculated within the calc clause are maintained in the result; for all the other Attributes that are defined as viral, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule section in the User Manual).

As mentioned, the Analytic invocation at component level can be done within the **calc** clause, both as part of a Join operator and the **calc** operator (see the parameter aggrCalc of those operators), therefore, for a better comprehension fo the behaviour at Component level, see also those operators.

Examples

Given the Data Set DS_1:

DS_r			
Id_1	**Id_2**	**Id_3**	**Me_1**
2010	E	XX	5
2010	B	XX	-3
2010	R	XX	9
2010	E	YY	13
2011	E	XX	11
2011	B	ZZ	7
2011	E	YY	-1
2011	F	XX	0
2012	L	ZZ	-2
2012	E	YY	3

Example1:

DS_r := sum (DS_1 over (order by Id_1, Id_2, Id_3 data points between 1 preceding and 1 following))

results in:

DS_r			
Id_1	**Id_2**	**Id_3**	**Me_1**

2010	B	XX	2
2010	E	XX	15
2010	E	YY	27
2010	R	XX	29
2011	B	ZZ	27
2011	E	XX	17
2011	E	YY	10
2011	F	XX	2
2012	E	YY	1
2012	L	ZZ	1

Counting the number of data points: count

Aggregate syntax

count (**dataset { groupingClause } **) *(in a Data Set expression)*

count (**component **) { groupingClause } *(in a Component expression within an* ***aggr*** *clause)*

count ( ) *(in an* ***having*** *clause)*

Analytic syntax

count (**dataset **over (analyticClause **))** *(in a Data Set expression)*

count (**component **over (analyticClause **))** *(in a Component expression within a* ***calc*** *clause)*

Input parameters

dataset the operand Data Set
component the operand Component
groupingClause see Aggregate invocation
analyticClause see Analytic invocation

Examples of valid syntaxes

See Aggregate and Analytic invocations above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset :: dataset
component :: component

Result type

result :: dataset { measure<integer> int_var }
| component<integer>

Additional constraints

None.

Behaviour

The operator returns the number of the input Data Points.
For other details, see Aggregate and Analytic invocations.

Examples

Given the Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
2011	A	XX	iii
2011	A	YY	jjj
2011	B	YY	iii
2012	A	XX	kkk
2012	B	YY	iii

Example 1: DS_r := count (DS_1 group by Id_1) results in:

DS_r	
Id_1	**Int_var**
2011	3
2012	2

Example 1: use of count in a **having** clause:

DS_r := sum (DS_1 group by Id_1 having count() > 2) results in:

DS_r	
Id_1	**Int_var**
2011	3

Minimum value : min

Aggregate syntax

min (** dataset { groupingClause } **) *(in a Data Set expression)*

min (** component **) { groupingClause } *(in a Component expression within an* **aggr** *clause)*

Analytic syntax

min (** dataset **over (analyticClause **))** *(in a Data Set expression)*

min (** component **over (analyticClause **))** *(in a Component expression within a* **calc** *clause)*

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Examples of valid syntaxes

See Aggregate and Analytic invocations above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type
dataset :: dataset
component :: component

Result type
result :: dataset
| component

Additional constraints
None.

Behaviour
The operator returns the minimum value of the input values.
For other details, see Aggregate and Analytic invocations.

Examples
Given the Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1: DS_r := min (DS_1 group by Id_1) results in:

DS_r	
Id_1	**Me_1**
2011	3
2012	2

Maximum value : **max**

Aggregate syntax
max (dataset { groupingClause } **)** *(in a Data Set expression)*
max (component) { groupingClause } *(in a Component expression within an* **aggr** *clause)*

Analytic syntax
max (dataset over (analyticClause **))** *(in a Data Set expression)*
max (component over (analyticClause **))** *(in a Component expression within a* **calc** *clause)*

Input parameters
dataset the operand Data Set
component the operand Component
groupingClause see Aggregate invocation
analyticClause see Analytic invocation

Examples of valid syntaxes

See Aggregate and Analytic invocations above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset :: dataset

component :: component

Result type

result :: dataset

| component

Additional constraints

None.

Behaviour

The operator returns the maximum of the input values.
For other details, see Aggregate and Analytic invocations.

Examples

Given the Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1: DS_r := max (DS_1 group by Id_1) results in:

DS_r	
Id_1	**Me_1**
2011	7
2012	4

Median value : **median**

Aggregate syntax

median **(** dataset { groupingClause } **)** *(in a Data Set expression)*

median **(** component **)** { groupingClause } *(in a Component expression within an* **aggr** *clause)*

Analytic syntax

median **(** dataset **over (** analyticClause **))** *(in a Data Set expression)*

median **(** component **over (** analyticClause **))** *(in a Component expression within a* **calc** *clause)*

Input parameters

dataset	the operand Data Set
component	the operand Component
<u>groupingClause</u>	see Aggregate invocation
<u>analyticClause</u>	see Analytic invocation

Examples of valid syntaxes

See Aggregate and Analytic invocations above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset :: dataset {measure<number>_+}
component :: component<number>

Result type

result :: dataset { measure<number> _+ }
| component<number>

Additional constraints

None.

Behaviour

The operator returns the median value of the input values.
For other details, see Aggregate and Analytic invocations.

Examples

Given the Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1: DS_r := median (DS_1 group by Id_1) results in:

DS_r	
Id_1	**Me_1**
2011	5
2012	3

Sum : sum

Aggregate syntax

sum **(** dataset { <u>groupingClause</u> } **)** *(in a Data Set expression)*

sum **(** component **)** { <u>groupingClause</u> } *(in a Component expression within an* **aggr** *clause)*

Analytic syntax

sum (dataset **over (** analyticClause **))** *(in a Data Set expression)*

sum (component **over (** analyticClause **))** *(in a Component expression within a* **calc** *clause)*

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Examples of valid syntaxes

See Aggregate and Analytic invocations above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset ::	dataset { measure<number> _+ }
component ::	component<number>

Result type

result ::	dataset { measure<number> _+ }
	| component<number>

Additional constraints

None.

Behaviour

The operator returns the sum of the input values.
For other details, see Aggregate and Analytic invocations.

Examples

Given the Data Set DS_1 :

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1: DS_r := sum (DS_1 group by Id_1) results in:

DS_r	
Id_1	**Me_1**
2011	15
2012	6

Average value : avg

Aggregate syntax

avg **(** dataset { groupingClause } **)** *(in a Data Set expression)*

avg **(** component **)** { groupingClause } *(in a Component expression within an* ***aggr*** *clause)*

Analytic syntax

avg **(** dataset **over (** analyticClause **))** *(in a Data Set expression)*

avg **(** component **over (** analyticClause **))** *(in a Component expression within a* ***calc*** *clause)*

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Examples of valid syntaxes

See Aggregate and Analytic invocations above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset :: dataset {measure<number> _+}
component :: component<number>

Result type

result :: dataset { measure<number> _+ }
| component<number>

Additional constraints

None.

Behaviour

The operator returns the mean of the input values.
For other details, see Aggregate and Analytic invocations.

Examples

Given the Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1: DS_r := avg (DS_1 group by Id_1) results in:

DS_r	
Id_1	**Me_1**
2011	5

2012	3

Population standard deviation : stddev_pop

Aggregate syntax

stddev_pop **(** dataset { groupingClause } **)** *(in a Data Set expression)*

stddev_pop **(** component **)** { groupingClause } *(in a Component expression within an* **aggr** *clause)*

Analytic syntax

stddev_pop **(** dataset **over (** analyticClause **))** *(in a Data Set expression)*

stddev_pop **(** component **over (** analyticClause **))** *(in a Component expression within a* **calc** *clause)*

Input parameters

dataset the operand Data Set
component the operand Component
groupingClause see Aggregate invocation
analyticClause see Analytic invocation

Examples of valid syntaxes

See Aggregate and Analytic invocations above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset :: dataset { measure<number> _+ }
component :: component<number>

Result type

result :: dataset { measure<number> _+ }
| component<number>

Additional constraints

None.

Behaviour

The operator returns the "population standard deviation" of the input values.
For other details, see Aggregate and Analytic invocations.

Examples

Given the Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1: DS_r := stddev_pop (DS_1 group by Id_1) results in:

DS_r	
Id_1	**Me_1**
2011	1.633
2012	1

Sample standard deviation : stddev_samp

Aggregate syntax

stddev_samp (**dataset { groupingClause } **) *(in a Data Set expression)*

stddev_samp (**component **) { groupingClause } *(in a Component expr. within an **aggr** clause)*

Analytic syntax

stddev_samp (**dataset **over (analyticClause **))** *(in a Data Set expression)*

stddev_samp (**component **over (analyticClause **))** *(in a Component expr. within a **calc** clause)*

Input parameters

dataset	the operand Data Set
component	the operand Component
groupingClause	see Aggregate invocation
analyticClause	see Analytic invocation

Semantics for scalar operations

This operator cannot be applied to scalar values.

Examples of valid syntaxes

See Aggregate and Analytic invocations above, at the beginning of the section.

Input parameters type

dataset ::	dataset { measure<number> _+ }
component ::	component<number>

Result type

result ::	dataset { measure<number> _+ }
	| component<number>

Additional constraints

None.

Behaviour

The operator returns the "sample standard deviation" of the input values.
For other details, see Aggregate and Analytic invocations.

Examples

Given the Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
2011	A	XX	3
2011	A	YY	5

2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1: DS_r := stddev_samp (DS_1 group by Id_1) results in:

DS_r	
Id_1	**Me_1**
2011	2
2012	1.4142

Population variance : var_pop

Aggregate syntax

var_pop (** dataset { groupingClause } **) *(in a Data Set expression)*

var_pop (** component **) { groupingClause } *(in a Component expression within an* **aggr** *clause)*

Analytic syntax

var_pop (** dataset **over (analyticClause **))** *(in a Data Set expression)*

var_pop (** component **over (analyticClause **))** *(in a Component expression within a* **calc** *clause)*

Input parameters

dataset the operand Data Set
component the operand Component
groupingClause see Aggregate invocation
analyticClause see Analytic invocation

Examples of valid syntaxes

See Aggregate and Analytic invocations above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset :: dataset {measure<number>_+}
component :: component<number>

Result type

result :: dataset { measure<number> _+ }
| component<number>

Additional constraints

None.

Behaviour

The operator returns the "population variance" of the input values.
For other details, see Aggregate and Analytic invocations.

Examples

Given the Data Set DS_1 :

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1: DS_r := var_pop (DS_1 group by Id_1) results in:

DS_r	
Id_1	**Me_1**
2011	2,6667
2012	1

Sample variance : var_samp

Aggregate syntax

var_samp **(** dataset { groupingClause } **)** *(in a Data Set expression)*

var_samp **(** component **)** { groupingClause } *(in a Component expression within an* **aggr** *clause)*

Analytic syntax

var_samp **(** dataset **over (** analyticClause **))** *(in a Data Set expression)*

var_samp **(** component **over (** analyticClause **))** *(in a Component expression within a* **calc** *clause)*

Input parameters

dataset the operand Data Set
component the operand Component
groupingClause see Aggregate invocation
analyticClause see Analytic invocation

Examples of valid syntaxes

See Aggregate and Analytic invocations above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset :: dataset {measure<number>_+}
component :: component<number>

Result type

result :: dataset { measure<number> _+ }
| component<number>

Additional constraints

None.

Behaviour

The operator returns the sample variance of the input values.

For other details, see Aggregate and Analytic invocations.

Examples

Given the Data Set DS_1

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
2011	A	XX	3
2011	A	YY	5
2011	B	YY	7
2012	A	XX	2
2012	B	YY	4

Example 1: DS_r := var_samp (DS_1 group by [Id_1]) results in:

DS_r	
Id_1	**Me_1**
2011	4
2012	2

First value : first_value

Syntax

first_value (dataset **over (** analyticClause **))** *(in a Data Set expression)*

first_value (component **over (** analyticClause **))** *(in a Component expression within a* **calc** *clause)*

Input parameters

dataset the operand Data Set
component the operand Component
analyticClause see Analytic invocation

Examples of valid syntaxes

See Analytic invocation above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset :: dataset { measure<scalar> _+ }
component :: component<scalar>

Result type

result :: dataset
| component<scalar>

Additional constraints

The Aggregate invocation is not allowed.

Behaviour

The operator returns the first value (in the value order) of the set of Data Points that belong to the same analytic window as the current Data Point.

When invoked at Data Set level, it returns the first value for each Measure of the input Data Set. The first value of different Measures can result from different Data Points.

When invoked at Component level, it returns the first value of the specified Component.

For other details, see Analytic invocation.

Examples

Given the Data Set DS_1 :

DS_1				
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**
A	XX	1993	3	1
A	XX	1994	4	9
A	XX	1995	7	5
A	XX	1996	6	8
A	YY	1993	9	3
A	YY	1994	5	4
A	YY	1995	10	2
A	YY	1996	2	7

Example 1:

DS_r := first_value (DS_1 over (partition by Id_1, Id_2 order by Id_3 data points between 1 preceding and 1 following))

results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**
A	XX	1993	3	1
A	XX	1994	3	1
A	XX	1995	4	5
A	XX	1996	6	5
A	YY	1993	5	3
A	YY	1994	5	2
A	YY	1995	2	2
A	YY	1996	2	2

Last value : last_value

Syntax

last_value (** dataset **over (analyticClause **))** *(in a Data Set expression)*

last_value (** component **over (analyticClause **))** *(in a Component expression within a* **calc** *clause)*

Input parameters

dataset the operand Data Set

component	the operand Component
analyticClause	see Analytic invocation

Examples of valid syntaxes

See Analytic invocation above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset ::	dataset {measure<scalar> _+}
component ::	component<scalar>

Result type

result ::	dataset
	| component<scalar>

Additional constraints

The Aggregate invocation is not allowed.

Behaviour

The operator returns the last value (in the value order) of the set of Data Points that belong to the same analytic window as the current Data Point.
When invoked at Data Set level, it returns the last value for each Measure of the input Data Set. The last value of different Measures can result from different Data Points.
When invoked at Component level, it returns the last value of the speficied Component.
For other details, see Analytic invocation.

Examples

Given the Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**
A	XX	1993	3	1
A	XX	1994	4	9
A	XX	1995	7	5
A	XX	1996	6	8
A	YY	1993	9	3
A	YY	1994	5	4
A	YY	1995	10	2
A	YY	1996	2	7

Example 1:

DS_r := last_value (DS_1 over (partition by Id_1, Id_2 order by Id_3 data points between 1 preceding and 1 following))

results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**
A	XX	1993	4	9

A	XX	1994	7	9
A	XX	1995	7	9
A	XX	1996	7	8
A	YY	1993	9	4
A	YY	1994	10	4
A	YY	1995	10	7
A	YY	1996	10	7

Lag : lag

Syntax

in a Data Set expression:
lag (** dataset {,** offset {**,** *defaultValue* } } **over (** { partitionClause } orderClause**))**

In a Component expression within a **calc** *clause:*
lag (** component {,** offset {**,** *defaultValue* } } **over (** { partitionClause } orderClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
offset	the relative position prior to the current Data Point
defaultValue	the value returned when the offset goes outside of the partition.
partitionClause	see Analytic invocation
orderClause	see Analytic invocation

Examples of valid syntaxes
See Analytic invocation above, at the beginning of the section.

Semantics for scalar operations
This operator cannot be applied to scalar values.

Input parameters type

dataset ::	dataset
component ::	component
offset ::	integer [value > 0]
default value ::	scalar

Result type

result ::	dataset
	| component

Additional constraints
The Aggregate invocation is not allowed.
The windowClause of the Analytic invocation syntax is not allowed.

Behaviour
In the ordered set of Data Points of the current partition, the operator returns the value(s) taken from the Data Point at the specified physical offset prior to the current Data Point.
If defaultValue is not specified then the value returned when the offset goes outside the partition is NULL.
For other details, see Analytic invocation.

Examples
Given the Data Set DS_1 :

DS_1				
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**
A	XX	1993	3	1
A	XX	1994	4	9
A	XX	1995	7	5
A	XX	1996	6	8
A	YY	1993	9	3
A	YY	1994	5	4
A	YY	1995	10	2
A	YY	1996	2	7

Example 1: DS_r := lag (DS_1 , 1 over (partition by Id_1 , Id_2 order by Id_3)) results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**
A	XX	1993	NULL	NULL
A	XX	1994	3	1
A	XX	1995	4	9
A	XX	1996	7	5
A	YY	1993	NULL	NULL
A	YY	1994	9	3
A	YY	1995	5	4
A	YY	1996	10	2

lead : lead

Syntax

in a Data Set expression:
lead (**dataset **, **{offset {,** defaultValue } } **over (** { partitionClause } orderClause **))**

in a Component expression within a **calc** *clause:*
lead (**component **, **{offset {,** defaultValue } } **over (** { partitionClause } orderClause **))**

Input parameters

dataset	the operand Data Set
component	the operand Component
offset	the relative position beyond the current Data Point
defaultValue	the value returned when the offset goes outide the partition.
partitionClause	see Analytic invocation
orderClause	see Analytic invocation

Examples of valid syntaxes
See Analytic invocation above, at the beginning of the section.

Semantics for scalar operations
This operator cannot be applied to scalar values.

Input parameters type

dataset :: dataset
component :: component
offset :: integer [value > 0]
default value :: scalar

Result type

result :: dataset
| component

Additional constraints

The Aggregate invocation is not allowed.
The windowClause of the Analytic invocation syntax is not allowed.

Behaviour

In the ordered set of Data Points of the current partition, the operator returns the value(s) taken from the Data Point at the specified physical offset beyond the current Data Point.
If defaultValue is not specified, then the value returned when the offset goes outside the partition is NULL.
For other details, see Analytic invocation.

Examples

Given the Data Set DS_1

DS_1				
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**
A	XX	1993	3	1
A	XX	1994	4	9
A	XX	1995	7	5
A	XX	1996	6	8
A	YY	1993	9	3
A	YY	1994	5	4
A	YY	1995	10	2
A	YY	1996	2	7

Example 1: DS_r := lead (DS_1 , 1 over (partition by Id_1 , Id_2 order by Id_3)) results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**
A	XX	1993	4	9
A	XX	1994	7	5
A	XX	1995	6	8
A	XX	1996	NULL	NULL
A	YY	1993	5	4
A	YY	1994	10	2
A	YY	1995	2	7
A	YY	1996	NULL	NULL

Rank : rank

Syntax

rank (over ({ partitionClause } orderClause **))** *(in a Component expression within a* **calc** *clause)*

Input parameters

partitionClause see Analytic invocation
orderClause see Analytic invocation

Examples of valid syntaxes

See Analytic invocation above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset :: dataset
component :: component

Result type

result :: dataset { measure<integer> int_var }
| component<integer>

Additional constraints

The invocation at Data Set level is not allowed.
The Aggregate invocation is not allowed.
The windowClause of the Analytic invocation syntax is not allowed.

Behaviour

The operator returns an order number (rank) for each Data Point, starting from the number 1 and following the order specified in the orderClause. If some Data Points are in the same order according to the specified orderClause, the same order number (rank) is assigned and a gap appears in the sequence of the assigned ranks (for example, if four Data Points have the same rank 5, the following assigned rank would be 9).
For other details, see Analytic invocation.

Examples

Given the Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**
A	XX	2000	3	1
A	XX	2001	4	9
A	XX	2002	7	5
A	XX	2003	6	8
A	YY	2000	9	3
A	YY	2001	5	4
A	YY	2002	10	2
A	YY	2003	5	7

Example 1:

DS_r := DS_1 [calc Me2 := rank (over (partition by Id_1 , Id_2 order by Me_1)) results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**
A	XX	2000	3	1
A	XX	2001	4	2
A	XX	2002	7	4
A	XX	2003	6	3
A	YY	2000	9	3
A	YY	2001	5	1
A	YY	2002	10	4
A	YY	2003	5	1

Ratio to report : ratio_to_report

Syntax

ratio_to_report **(** dataset **over (** partitionClause **))** *(in a Data Set expression)*

ratio_to_report **(** component **over (** partitionClause **))** *(in a Component expr. within a* ***calc*** *clause)*

Input parameters

dataset — the operand Data Set
component — the operand Component
partitionClause — see Analytic invocation

Examples of valid syntaxes

See Analytic invocation above, at the beginning of the section.

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type

dataset :: dataset { measure<number>_+ }
component :: component<number>

Result type

result :: dataset { measure<number> _+ }
| component<number>

Additional constraints

The Aggregate invocation is not allowed.
The orderClause and windowClause of the Analytic invocation syntax are not allowed.

Behaviour

The operator returns the ratio between the value of the current Data Point and the sum of the values of the partition which the current Data Point belongs to.
For other details, see Analytic invocation.

Examples

Given the Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**
A	XX	2000	3	1

A	XX	2001	4	3
A	XX	2002	7	5
A	XX	2003	6	1
A	YY	2000	12	0
A	YY	2001	8	8
A	YY	2002	6	5
A	YY	2003	14	-3

Example 1: DS_r := ratio_to_report (DS_1 over (partition by Id_1, Id_2)) results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**
A	XX	2000	0.15	0,1
A	XX	2001	0.2	0.3
A	XX	2002	0.35	0.5
A	XX	2003	0.3	0.1
A	YY	2000	0.3	0
A	YY	2001	0.2	0.8
A	YY	2002	0.15	0.5
A	YY	2003	0.35	-0.3

VTL-ML - Data validation operators

check_datapoint

Syntax

check_datapoint (op **,** dpr { **components** listComp } { **output** output } **)**

listComp ::= comp { **,** comp }*

output ::= **invalid | all | all_measures**

Input parameters

op	the Data Set to check
dpr	the Data Point Ruleset to be used
listComp	if dpr is defined on Value Domains then listComp is the list of Components of op to be associated (in positional order) to the conditioning Value Domains defined in dpr. If dpr is defined on Variables then listComp is the list of Components of op to be associated (in positional order) to the conditioning Variables defined in dpr (for documentation purposes).
comp	Component of op
output	specifies the Data Points and the Measures of the resulting Data Set:

invalid	the resulting Data Set contains a Data Point for each Data Point of op and each Rule in dpr that evaluates to FALSE on that Data Point. The resulting Data Set has the Measures of op.
all	the resulting Data Set contains a data point for each Data Point of op and each Rule in dpr. The resulting Data Set has the *boolean* Measure bool_var.
all_measures	the resulting Data Set contains a Data Point for each Data Point of op and each Rule in dpr. The resulting dataset has the Measures of op and the *boolean* Measure bool_var.

If not specified then output is assumed to be invalid. See the Behaviour for further details.

Examples of valid syntaxes

check_datapoint (DS1, DPR invalid)
check_datapoint (DS1, DPR all_measures)

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type:

op :: dataset
dpr :: name < datapoint >
comp :: name < component >

Result type:

result :: dataset

Additional constraints

If dpr is defined on Value Domains then it is mandatory to specify listComp. The Components specified in listComp must belong to the operand op and be defined on the Value Domains specified in the signature of dpr.
If dpr is defined on Variables then the Components specified in the signature of dpr must belong to the operand op.
If dpr is defined on Variables and listComp is specified then the Components specified in listComp are the same, in the same order, as those specified in op (they are provided for documentation purposes).

Behaviour

It returns a Data Set having the following Components:

- the Identifier Components of op
- the Identifier Component ruleid whose aim is to identify the Rule that has generated the actual Data Point (it contains at least the Rule name specified in dpr [8])
- if the output parameter is **invalid**: the original Measures of op (no *boolean* measure)
- if the output parameter is **all**: the *boolean* Measure bool_var whose value is the result of the evaluation of a rule on a Data Point (TRUE, FALSE or NULL).
- if the output parameter is **all_measures**: the original measures of op and the *boolean* Measure bool_var whose value is the result of the evaluation of a rule on a Data Point (TRUE, FALSE or NULL).
- the Measure errorcode that contains the errorcode specified in the rule
- the Measure errorlevel that contains the errorlevel specified in the rule

A Data Point of op can produce several Data Points in the resulting Data Set, each of them with a different value of ruleid. If output is **invalid** then the resulting Data Set contains a Data Point for each Data Point of op and each rule of dpr that evaluates to FALSE. If output is **all** or **all_measures** then the resulting Data Set contains a Data Point for each Data Point of op and each rule of dpr.

Examples

```
define datapoint ruleset dpr1 ( variable Id_3, Me_1 ) is
        when Id_3 = “CREDIT” then Me_1 >= 0 errorcode “Bad credit”
      ; when Id_3 = “DEBIT” then Me_1 >= 0 errorcode “Bad debit”
end datapoint ruleset
```

Given the Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
2011	I	CREDIT	10
2011	I	DEBIT	-2
2012	I	CREDIT	10
2012	I	DEBIT	2

DS_r := check_datapoint (DS_1, dpr1)   results in:

DS_r						
Id_1	**Id_2**	**Id_3**	**ruleid**	**obs_value**	**errorcode**	**errorlevel**
2011	I	DEBIT	dpr1_2	-2	Bad debit	

DS_r := check_datapoint (DS_1, dpr1 all)   results in:

DS_r						
Id_1	**Id_2**	**Id_3**	**ruleid**	**bool_var**	**errorcode**	**errorlevel**
2011	I	CREDIT	dpr1_1	true		
2011	I	CREDIT	dpr1_2	true		
2011	I	DEBIT	dpr1_1	true		

[8] The content of ruleid maybe personalised in the implementation

2011	I	DEBIT	dpr1_2	false	Bad debit	
2012	I	CREDIT	dpr1_1	true		
2012	I	CREDIT	dpr1_2	true		
2012	I	DEBIT	dpr1_1	true		
2012	I	DEBIT	dpr1_2	true		

check_hierarchy

Syntax

check_hierarchy (op , hr { **condition** condComp { **,** condComp }* } { **rule** ruleComp }
{ mode } { input } { output } **)**

mode ::= **non_null | non_zero | partial_null | partial_zero | always_null | always_zero**

input ::= **dataset | dataset_priority**

output ::= **invalid | all | all_measures**

Input parameters

op — the Data Set to be checked

hr — the hierarchical Ruleset to be used

condComp — condComp is a Component of op to be associated (in positional order) to the conditioning Value Domains or Variables defined in hr (if any).

ruleComp — ruleComp is the Identifier of op to be associated to the rule Value Domain or Variable defined in hr.

mode — this parameter specifies how to treat the possible missing Data Points corresponding to the Code Items in the left and right sides of the rules and which Data Points are produced in output. The meaning of the possible values of the parameter is explained below.

input — this parameter specifies the source of the values used as input of the comparisons. The meaning of the possible values of the parameter is explained below.

output — this parameter specifies the structure and the content of the resulting dataset. The meaning of the possible values of the parameter is explained below.

Examples of valid syntaxes

check_hierarchy (DS1, HR_2 non_null dataset invalid)
check_hierarchy (DS1, HR_3 non_zero dataset_priority all)

Input parameters type

op :: dataset { measure<number> _ }
hr :: name < hierarchical >
condComp :: name < component >
ruleComp :: name < identifier >

Result type

result :: dataset {measure<number> _ }

Additional constraints

If hr is defined on Value Domains then it is mandatory to specify the condition (if any in the ruleset hr) and the rule parameters. Moreover, the Components specified as condComp and ruleComp must belong to the operand op and must take values on the Value Domains corresponding, in positional order, to the ones specified in the condition and rule parameter of hr.

If hr is defined on Variables, the specification of condComp and ruleComp is not needed, but they can be specified all the same if it is desired to show explicitly in the invocation which are the involved Components: in this case, the condComp and ruleComp must be the same and in the same order as the Variables specified in in the condition and rule signatures of hr.

Behaviour

The **check_hierarchy** operator applies the Rules of the Ruleset hr to check the Code Items Relations between the Code Items present in op (as for the Code Items Relations, see the User Manual - section "Generic Model for Variables and Value Domains"). The operator checks if the relation between the left and the right member is fulfilled, giving TRUE in positive case and FALSE in negative case.

The Attribute propagation rule is applied on each group of Data Points which contributes to the same Data Point of the result.

The behaviours relevanto to the different options of the input parameters are the following.

First, the parameter input is used to determine the source of the Data Points used as input of the check_hierarchy. The possible options of the parameter input and the corresponding behaviours are the following:

dataset — this option addresses the case where all the input Data Points of all the Rules of the Ruleset are expected to be taken from the input Data Set (the operand op).
For each Rule of the Ruleset and for each item on the left and right sides of the Rule, the operator takes the input Data Points exclusively from the operand op.

dataset_prority — this option addresses the case where the input Data Points of all the Rules of the Ruleset are preferably taken from the input Data Set (the operand op), however if a valid Measure value for an expected Data Point is not found in op, the attempt is made to take it from the computed output of a (possible) other Rule.
For each Rule of the Ruleset and for each item on the left and right sides of the Rule:

- if the item is not defined as the result (left side) of another Rule that applies the Code Item relation "is equal to" (=), the current Rule takes the input Data Points from the operand op.
- if the item is defined as result of another Rule R that applies the Code Item relation "is equal to" (=), then:
  - if an expected input Data Point exists in op and its Measure is <u>not</u> NULL, then the current Rule takes such Data Point from op;
  - if an expected input Data Point does not exist in op or its measure is NULL, then the current Rule takes the Data Point (if any) that has the same Identifiers' values from the computed output of the other Rule R;

if the parameter input is not specified then it is assumed to be dataset.

Then the parameter mode is considered, to determine the behaviour for missing Data Points and for the Data Points to be produced in the output. The possible options of the parameter mode and the corresponding behaviours are the following:

non_null — the result Data Point is produced when all the items involved in the comparison exist and have not NULL Measure value (i.e., when no Data Point corresponding to the Code Items of the left and right sides of the rule is missing or has NULL Measure value); under this option, in evaluating the comparison, the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a NULL Measure value;

non_zero — the result Data Point is produced when at least one of the items involved in the comparison exist and have Measure not equal to 0 (zero); the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a Measure value equal to 0;

partial_null — the result Data Point is produced if at least one Data Point corresponding to the Code Items of the left and right sides of the rule is found (whichever is its Measure value); the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a NULL Measure value;

partial_zero — the result Data Point is produced if at least one Data Point corresponding to the Code Items of the left and right sides of the rule is found (whichever is its Measure value); the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a Measure value equal to 0 (zero);

always_null — the result Data Point is produced in any case; the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a Measure value equal to NULL;

always_zero — the result Data Point is produced in any case; the possible missing Data Points corresponding to the Code Items of the left and right sides of the rule are considered existing and having a Measure value equal to 0 (zero);

If the parameter mode is not specified, then it is assumed to be non_null.

The following table summarizes the behaviour of the options of the parameter "mode"

OPTION of the MODE PARAMETER:	Missing Data Points are considered:	Null Data Points are considered:	Condition for evaluating the rule	Returned Data Points
Non_null	NULL	NULL	If all the involved Data Points are not NULL	Only not NULL Data Points (Zeros are returned too)
Non_zero	Zero	NULL	If at least one of the involved Data Points is <> zero	Only not zero Data Points (NULLS are returned too)
Partial_null	NULL	NULL	If at least one of the involved Data Points is not NULL	Data Points of any value (NULL, not NULL and zero too)
Partial_zero	Zero	NULL	If at least one of the involved Data Points is not NULL	Data Points of any value (NULL, not NULL and zero too)
Always_null	NULL	NULL	Always	Data Points of any value (NULL, not NULL and zero too)
Always_zero	Zero	NULL	Always	Data Points of any value (NULL, not NULL and zero too)

Finally the parameter output is considered, to determine the structure and content of the resulting Data Set. The possible options of the parameter output and the corresponding behaviours are the following:

all — all the Data Points produced by the comparison are returned, both the valid ones (TRUE) and the invalid ones (FALSE) besides the possible NULL ones. The result of the comparison is returned in the *boolean* Measure bool_var. The original Measure Component of the Data Set op is not returned.

invalid — only the invalid (FALSE) Data Points produced by the comparison are returned. The result of the comparison (*boolean* Measure bool_var) is not returned. The original Measure Component of the Data Set op is returned and contains the Measure values taken from the Data Points on the left side of the rule.

all_measures — all the Data Points produced by the comparison are returned, both the valid ones (TRUE) and the invalid ones (FALSE) besides the possible NULL ones. The result of the comparison is returned in the *boolean* Measure bool_var. The original Measure Component of the Data Set op is returned and contains the Measure values taken from the Data Points on the left side of the rule.

If the parameter output is not specified then it is assumed to be invalid.

In conclusion, the operator returns a Data Set having the following Components:

- all the Identifier Components of op
- the additional Identifier Component ruleid, whose aim is to identify the Rule that has generated the actual Data Point (it contains at least the Rule name specified in hr [9])
- if the output parameter is all: the *boolean* Measure bool_var whose values are the result of the evaluation of the Rules (TRUE, FALSE or NULL).
- if the output parameter is invalid: the original Measure of op, whose values are taken from the Measure values of the Data Points of the left side of the Rule
- if the output parameter is all_measures: the *boolean* Measure bool_var, whose value is the result of the evaluation of a Rule on a Data Point (TRUE, FALSE or NULL), and the original Measure of op, whose values are taken from the Measure values of the Data Points of the left side of the Rule
- the Measure imbalance, which contains the difference between the Measure values of the Data Points on the left side of the Rule and the Measure values of the corresponding calculated Data Points on the right side of the Rule
- the Measure errorcode, which contains the errorcode value specified in the Rule
- the Measure errorlevel, which contains the errorlevel value specified in the Rule

Note that a gereric Data Point of op can produce several Data Points in the resulting Data Set, one for each Rule in which the Data Point appears as the left member of the comparison.

Examples

See also the examples in **define hierarchical ruleset**.

Given the following hierarchical ruleset:

```
define hierarchical ruleset  HR_1 ( valuedomain rule  VD_1 )  is
          R010 :     A = J + K + L                    errorlevel 5
        ; R020 :     B = M + N + O                    errorlevel 5
        ; R030 :     C = P + Q        errorcode XX    errorlevel 5
        ; R040 :     D = R + S                        errorlevel 1
        ; R060 :     F = Y + W + Z                    errorlevel 7
        ; R070 :     G = B + C
        ; R080 :     H = D + E                        errorlevel 0
        ; R090 :     I  = D + G       errorcode YY    errorlevel 0
        ; R100 :     M  >=  N                         errorlevel 5
        ; R110 :     M  <=  G                         errorlevel 5
end hierarchical ruleset
```

And given the operand Data Set DS_1 (where At_1 is viral and the propagation rule says that the alphabetic order prevails the NULL prevails on the alphabetic characters and the Attribute value for missing Data Points is assumed as NULL):

DS_1		
Id_1	**Id_2**	**Me_1**
2010	A	5
2010	B	11
2010	C	0
2010	G	19
2010	H	NULL

[9] The content of ruleid maybe personalised in the implementation

2010	I	14
2010	M	2
2010	N	5
2010	O	4
2010	P	7
2010	Q	-7
2010	S	3
2010	T	9
2010	U	NULL
2010	V	6

Example 1: DS_r := check_hierarchy (DS_1, HR_1 rule Id_2 partial_null all) results in:

DS_r						
Id_1	**Id_2**	**ruleid**	**Bool_var**	**imbalance**	**errorcode**	**errorlevel**
2010	A	R010	NULL	NULL	NULL	5
2010	B	R020	TRUE	0	NULL	5
2010	C	R030	TRUE	0	XX	5
2010	D	R040	NULL	NULL	NULL	1
2010	E	R050	NULL	NULL	NULL	0
2010	F	R060	NULL	NULL	NULL	7
2010	G	R070	FALSE	8	NULL	NULL
2010	H	R080	NULL	NULL	NULL	0
2010	I	R090	NULL	NULL	YY	0
2010	M	R100	FALSE	-3	NULL	5
2010	M	R110	TRUE	-17	NULL	5

check

Syntax

check (op { **errorcode** errorcode } { **errorlevel** errorlevel } { **imbalance** imbalance } { output } **)**

output ::= **invalid | all**

Input parameters

op a *boolean* Data Set (a *boolean* condition expressed on one or more Data Sets)

errorcode the error code to be produced when the condition evaluates to FALSE. It must be a valid value of the errorcode_vd Value Domain (or *string* if the errorcode_vd Value Domain is not found). It can be a Data Set or a *scalar*. If not specified then errorcode is NULL.

errorlevel the error level to be produced when the condition evaluates to FALSE. It must be a valid value of the errorlevel_vd Value Domain (or *integer* if the errorcode_vd Value Domain is not found). It can be a Data Set or a *scalar*. If not specified then errorlevel is NULL.

imbalance — the imbalance to be computed. imbalance is a *numeric* mono-measure Data Set with the same Identifiers of op. If not specified then imbalance is NULL.

output — specifies which Data Points are returned in the resulting Data Set:

- **invalid** — returns the Data Points of op for which the condition evaluates to FALSE
- **all** — returns all Data Points of op

If not specified then output is **all**.

Examples of valid syntaxes

check (DS1 > DS2 errorcode myerrorcode errorlevel myerrorlevel imbalance DS1 - DS2 invalid)

Input parameters type:

op :: dataset
errorcode :: errorcode_vd
errorlevel :: errorlevel_vd
imbalance :: number

Result type:

result :: dataset

Additional constraints

op has exactly a *boolean* Measure Component.

Behaviour

It returns a Data Set having the following components:

- the Identifier Components of op
- a *boolean* Measure named **bool_var** that contains the result of the evaluation of the *boolean* dataset op
- the Measure imbalance that contains the specified imbalance
- the Measure errorcode that contains the specified errorcode
- the Measure errorlevel that contains the specified errorlevel

If output is **all** then all data points are returned. If output is **invalid** then only the Data Points where bool_var is FALSE are returned.

Examples

Given the Data Sets DS_1 and DS_2 :

DS_1		
Id_1	Id_2	Me_1
2010	I	1
2011	I	2
2012	I	10
2013	I	4
2014	I	5
2015	I	6
2010	D	25
2011	D	35
2012	D	45

2013	D	55
2014	D	50
2015	D	75

DS_2		
Id_1	Id_2	Me_1
2010	I	9
2011	I	2
2012	I	10
2013	I	7
2014	I	5
2015	I	6
2010	D	50
2011	D	35
2012	D	40
2013	D	55
2014	D	65
2015	D	75

Example 1: DS_r := check (DS1 >= DS2 imbalance DS1 - DS2) returns:

DS_r					
Id_1	Id_2	bool_var	imbalance	errorcode	errorlevel
2010	I	FALSE	-8	NULL	NULL
2011	I	TRUE	0	NULL	NULL
2012	I	TRUE	0	NULL	NULL
2013	I	FALSE	-3	NULL	NULL
2014	I	TRUE	0	NULL	NULL
2015	I	TRUE	0	NULL	NULL
2010	D	FALSE	-25	NULL	NULL
2011	D	TRUE	0	NULL	NULL
2012	D	TRUE	5	NULL	NULL
2013	D	TRUE	0	NULL	NULL
2014	D	FALSE	-15	NULL	NULL
2015	D	TRUE	0	NULL	NULL

VTL-ML - Conditional operators

if-then-else : if

Syntax

if condition **then** thenOperand **else** elseOperand

Input parameters

condition a Boolean condition (dataset, component or scalar)
thenOperand the operand returned when condition evaluates to **true**
elseOperand the operand returned when condition evaluates to **false**

Examples of valid syntaxes

if A > B then A else B

Semantics for scalar operations

The **if** operator returns thenOperand if condition evaluates to **true**, elseOperand otherwise. For example, considering the statement:

if x1 > x2 then 2 else 5,
for x1 = 3, x2 =0 it returns 2
for x1 = 0, x2 =3 it returns 5

Input Parameters type

condition :: dataset { measure <boolean> _ }
| component<Boolean>
| boolean
thenOperand :: dataset
| component
| scalar
elseOperand :: dataset
| component
| scalar

Result type

result :: dataset
| component<
| scalar

Additional constraints

- The operands thenOperand and elseOperand must be of the same scalar type.
- If the operation is at scalar level, thenOperand and elseOperand are scalar then condition must be scalar too (a *boolean* scalar).
- If the operation is at Component level, at least one of thenOperand and elseOperand is a Component (the other one can be scalar) and condition must be a Component too (a *boolean* Component); thenOperand, elseOperand and the other Components referenced in condition must belong to the same Data Set.
- If the operation is at Data Set level, at least one of thenOperand and elseOperand is a Data Set (the other one can be scalar) and condition must be a Data Set too (having a unique *boolean* Measure) and must have the same Identifiers as thenOperand or/and ElseOperand
    - If thenOperand and elseOperand are both Data Sets then they must have the same Components in the same roles
    - If one of thenOperand and elseOperand is a Data Set and the other one is a scalar, the Measures of the operand Data Set must be all of the same scalar type as the scalar operand.

Behaviour

For operations at Component level, the operation is applied for each Data Point of the unique input Data Set, the **if-then-else** operator returns the value from the thenOperand Component when condition evaluates to **true**, otherwise it returns the value from the elseOperand Component. If one of the operands thenOperand or elseOperand is scalar, such a scalar value can be returned depending on the outcome of the condition.

For operations at Data Set level, the **if-then-else** operator returns the Data Point from thenOperand when the Data Point of condition having the same Identifiers' values evaluates to **true**, and returns the Data Point from elseOperand otherwise. If one of the operands thenOperand or elseOperand is scalar, such a scalar value can be returned (depending on the outcome of the condition) and in this case it feeds the values of all the Measures of the result Data Point.

The behaviour for two Data Sets can be procedurally explained as follows. First the condition Data Set is evaluated, then its true Data Points are inner joined with thenOperand and its false Data Points are inner joined with elseOperand, finally the union is made of these two partial results (the condition ensures that there cannot be conflicts in the union).

Examples

Example 1: given the operand Data Sets DS_cond, DS_1, DS_2 :

DS_cond

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	M	5451780
2012	B	Total	F	5643070
2012	G	Total	M	5449803
2012	G	Total	F	5673231
2012	S	Total	M	23099012
2012	S	Total	F	23719207
2012	F	Total	M	31616281
2012	F	Total	F	33671580
2012	I	Total	M	28726599
2012	I	Total	F	30667608
2012	A	Total	M	NULL
2012	A	Total	F	NULL

DS_1

Id_1	Id_2	Id_3	Id_4	Me_1
2012	S	Total	F	25.8
2012	F	Total	F	NULL
2012	I	Total	F	20.9
2012	A	Total	M	6.3

DS_2

Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	M	0.12
2012	G	Total	M	22.5
2012	S	Total	M	23.7
2012	A	Total	F	NULL

DS_r := if (DS_cond#Id_4 = "F") then DS_1 else DS_2 returns:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	S	Total	F	25.8
2012	F	Total	F	NULL
2012	I	Total	F	20.9

Nvl : nvl

Syntax

nvl (op1 **,** op2 **)**

Input parameters

op1 the first operand
op2 the second operand

Examples of valid syntaxes

nvl (ds1#m1, 0)

Semantics for scalar operations

The operator nvl returns op2 when op1 is **null**, otherwise op1. For example:

nvl (5, 0) returns 5
nvl (null, 0) returns 0

Input Parameters type

op1 :: dataset
| component<scalar>
| scalar

op2 :: dataset
| component
| <scalar>

Result type

result :: dataset
| component
| scalar

Additional constraints

If op1 and op2 are scalar values then they must be of the same type.
If op1 and op2 are Components then they must be of the same type.
If op1 and op2 are Data Sets then they must have the same Components.

Behaviour

The operator nvl returns the value from op2 when the value from op1 is null, otherwise it returns the value from op1.
The operator has the typical behaviour of the operators applicable on two scalar values or Data Sets or Data Set Components.
Also the following statement gives the same result: if isnull (op1) then op2 else op1

Examples

Example 1: Given the input Data Set DS_1

DS_1				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	11094850
2012	G	Total	Total	11123034
2012	S	Total	Total	NULL
2012	M	Total	Total	417546
2012	F	Total	Total	5401267
2012	N	Total	Total	NULL

DS_r := nvl (DS_1, 0) returns:

DS_r				
Id_1	**Id_2**	**Id_3**	**Id_4**	**Me_1**
2012	B	Total	Total	11094850
2012	G	Total	Total	11123034
2012	S	Total	Total	0
2012	M	Total	Total	417546
2012	F	Total	Total	5401267
2012	N	Total	Total	0

VTL-ML - Clause operators

Filtering Data Points : filter

Syntax

op **[filter** filterCondition **]**

Input parameters

op the operand
filterCondition the filter condition

Examples of valid syntaxes

DS_1 [filter Me_3 > 0]
DS_1 [filter Me_3 + Me_2 <= 0]

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type:

op :: dataset
filterCondition :: component<boolean>

Result type:

result :: dataset

Additional constraints:

None.

Behaviour

The operator takes as input a Data Set (op) and a *boolean* Component expression (filterCondition) and filters the input Data Points according to the evaluation of the condition. When the expression is TRUE the Data Point is kept in the result, otherwise it is not kept (in other words, it filters out the Data Points of the operand Data Set for which filterCondition condition evaluates to FALSE or NULL).

Examples

Given the Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Me_1**	**At_1**
1	A	XX	2	E
1	A	YY	2	F
1	B	XX	20	F
1	B	YY	1	F
2	A	XX	4	E
2	A	YY	9	F

Example1: DS_r := DS_1 [filter Id_1 = 1 and Me_1 < 10] results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Me_1**	**At_1**

1	A	XX	2	E
1	A	YY	2	F
1	B	YY	1	F

Calculation of a Component : **calc**

Syntax

op **[calc** { <u>calcRole</u> } calcComp **:=** calcExpr { **,** { <u>calcRole</u> } calcComp **:=** calcExpr }* **]**

<u>calcRole</u> ::= **identifier** | **measure** | **attribute** | **viral attribute**

Input parameters

op — the operand
<u>calcRole</u> — the role to ba assigned to a Component to be calculated
calcComp — the name of a Component to be calculated
calcExpr — expression at component level, having only Components of the input Data Sets as operands, used to calculate a Component

Examples of valid syntaxes

DS_1 [calc Me_3 := Me_1 + Me_2]

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type:

op :: dataset
calcComp :: name < component >
calcExpr :: component<scalar>

Result type:

result :: dataset

Additional constraints

The calcComp parameter cannot be the name of an Identifier component.
All the components used in calcComp must belong to the operand Data Set op.

Behaviour

The operator calculates new Identifier, Measure or Attribute Components on the basis of sub-expressions at Component level. Each Component is calculated through an independent sub-expression. It is possible to specify the role of the calculated Component among **measure**, **identifier**, **attribute**, or **viral attribute**, therefore the calc clause can be used also to change the role of a Component when possible. The keyword **viral** allows controlling the virality of the calculated Attributes (for the attribute propagation rule see the User Manual). When the role is omitted, the following rule is applied: if the component exists in the operand Data Set then it maintains its role; if the component does not exist in the operand Data Set then its role is Measure.
The calcExpr sub-expressions are independent one another, they can only reference Components of the input Data Set and cannot use Components generated, for example, by other calcExpr. If the calculated Component is a new Component, it is added to the output Data Set. If the Calculated component is a Measure or an Attribute that already exists in the input Data Set, the calculated values overwrite the original values. If the calculated Component is an Identifier that already exists in the input Data Set, an exception is raised because overwriting an Identifier Component is forbidden for preserving the functional behaviour. Analytic invocations can be used in the **calc** clause.

Examples

Given the Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
1	A	CA	20
1	B	CA	2
2	A	CA	2

Example1: DS_r := DS_1 [calc Me_1:= Me_1 * 2] results in:

DS_r			
Id_1	**Id_2**	**Id_3**	**Me_1**
1	A	CA	40
1	B	CA	4
2	A	CA	4

Example2: DS_r := DS_1 [calc attribute At_1:= “EP”] results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Me_1**	**At_1**
1	A	CA	40	EP
1	B	CA	4	EP
2	A	CA	4	EP

Aggregation : aggr

Syntax

op **[aggr** <u>aggrClause</u> { <u>groupingClause</u> } **]**

<u>aggrClause</u> ::= { <u>aggrRole</u> } aggrComp **:=** aggrExpr
{ **,** { <u>aggrRrole</u> } aggrComp**:=** aggrExpr }*

<u>groupingClause</u> ::= { **group by** groupingId {, gropuingId }*
| **group except** groupingId {, groupingId }*
| **group all** conversionExpr }[1]
{ **having** havingCondition }

<u>aggrRole</u>::= **measure** | **attribute** | **viral attribute**

Input Parameters

op	the operand
<u>aggrClause</u>	clause that specifies the required aggregations, i.e., the aggregated Components to be calculated, their roles and their calculation algorithm, to be applied on the joined and filtered Data Points
<u>aggrRole</u>	the role of the aggregated Component to be calculated
aggrComp	the name of the aggregated Component to be calculated; this is a dependent Component of the result (Measure or Attribute, not Identifier)

aggrExpr — expression at component level, having only Components of the input Data Sets as operands, which invokes an aggregate operator (e.g. **avg**, **count**, **max** ... , see also the corresponding sections) to perform the desired aggregation. Note that the **count** operator is used in an aggrClause without parameters, e.g.:

DS_1 [aggr Me_1 := count ( ) group by Id_1)]

groupingClause — the following alternative grouping options:

- **group by** — the Data Points are grouped by the values of the specified Identifiers (groupingId). The Identifiers not specified are dropped in the result.
- **group except** — the Data Points are grouped by the values of the Identifiers not specified as groupingId. The Identifiers specified as groupingId are dropped in the result.
- **group all** — converts the values of an Identifier Component using conversionExpr and keeps all the resulting Identifiers.

groupingId — Identifier Component to be kept (in the **group by** clause) or dropped (in the **group except** clause).

conversionExpr — specifies a conversion operator (e.g., **time_agg**) to convert an Identifier from finer to coarser granularity. The conversion operator is applied on an Identifier of the operand Data Set op.

havingCondition — a condition (boolean expression) at component level, having only Components of the input Data Sets as operands (and possibly constants), to be fulfilled by the groups of Data Points: only groups for which havingCondition evaluates to TRUE appear in the result. The havingCondition refers to the groups specified through the groupingClause, therefore it must invoke aggregate operators (e.g. **avg**, **count**, **max** ..., see also the section Aggregate invocation). A correct example of havingCondition is:

max(obs_value) < 1000

instead the condition obs_value < 1000 is not a right havingCondition, because it refers to the values of the single Data Points and not to the groups. The **count** operator is used in a havingCondition without parameters, e.g.:

sum (DS_1 group by id1 having count ( ) >= 10)

Examples of valid syntaxes

DS_1 [aggr M1 := min (Me_1) group by Id_1, Id_2]
DS_1 [aggr M1 := min (Me_1) group except Id_1, Id_2]

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type:

op :: dataset
aggrComp :: name < component >
aggrExpr :: component<scalar>
groupingId :: name <identifier >
conversionExpr :: identifier<scalar>
havingCondition :: component<boolean>

Result type:

result :: dataset

Additional constraints

The aggrComp parameter cannot be the name of an Identifier component.
All the components used in aggrExpr must belong to the operand Data Set op.
The conversionExpr parameter applies just one conversion operator to just one Identifier belonging to the input Data Set. The basic scalar type of the Identifier must be compatible with the basic scalar type of the conversion operator.

Behaviour

The operator **aggr** calculates aggregations of dependent Components (Measures or Attributes) on the basis of sub-expressions at Component level. Each Component is calculated through an independent sub-expression. It is possible to specify the role of the calculated Component among **measure attribute**, or **viral attribute**. The substring **viral** allows to control the virality of Attributes, if the Attribute propagation rule is adopted (see the User Manual). When the role is omitted, the following rule is applied: if the component exists in the operand Data Set then it maintains its role; if the component does not exist in the operand Data Set then its role is Measure.
The aggrExpr sub-expressions are independent of one another, they can only reference Components of the input Data Set and cannot use Components generated, for example, by other aggrExpr sub-expressions. The **aggr** computed Measures and Attributes are the only Measures and Attributes returned in the output Data Set (plus the possible viral Attributes). The sub-expressions must contain only Aggregate operators, which are able to compute an aggregated Value relevant to a group of Data Points. The groups of Data Points to be aggregated are specified through the groupingClause, which allows the following alternative options.

group by the Data Points are grouped by the values of the specified Identifiers. The Identifiers not specified are dropped in the result.

group except the Data Points are grouped by the values of the Identifiers not specified in the clause. The specified Identifiers are dropped in the result.

group all converts an Identifier Component using conversionExpr and keeps all the other Identifiers.

The **having** clause is used to filter groups in the result by means of an aggregate condition evaluated on the single groups (for example the minimum number of Data Points in the group).
If no grouping clause is specified, then all the input Data Points are aggregated in a single group and the clause returns a Data Set that contains a single Data Point and has no Identifiers.
The Attributes calculated through the **aggr** clauses are maintained in the result. For all the other Attributes that are defined as **viral**, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule section in the User Manual).

Examples

Given the Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Id_3**	**Me_1**
1	A	XX	0
1	A	YY	2
1	B	XX	3
1	B	YY	5
2	A	XX	7
2	A	YY	2

Example1: DS_r := DS_1 [aggr Me_1:= sum(Me_1) group by Id_1 , Id_2] results in:

DS_r		
Id_1	**Id_2**	**Me_1**
1	A	2
1	B	8
2	A	9

Example2: DS_r := DS_1 [aggr Me_3:= min(Me_1) group except Id_3] results in:

DS_r		
Id_1	**Id_2**	**Me_3**
1	A	0
1	B	3
2	A	2

Example3: DS_r := DS_1 [aggr Me_1:= sum(Me_1), Me_2 := max(Me_1)
group by Id_1 , Id_2
having mean (Me_1) > 2] results in:

DS_r			
Id_1	**Id_2**	**Me_1**	**Me_2**
1	B	8	5
2	A	9	7

Maintaining Components: **keep**

Syntax

op **[keep** comp {**,** comp }* **]**

Input parameters

op the operand
comp a component to keep

Examples of valid syntaxes

DS_1 [keep Me_2, Me_3]

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type:

op :: dataset
comp :: name < component >

Result type:

result :: dataset

Additional constraints:

All the Components comp must belong to the input Data Set op.
The Components comp cannot be Identifiers in op.

Behaviour

The operator takes as input a Data Set (op) and some Component names of such a Data Set (comp). These Components can be Measures or Attributes of op but not Identifiers. The operator maintains the specified Components, drops all the other dependent Components of the Data Set (Measures and Attributes) and maintains the independent Components (Identifiers) unchanged. This operation corresponds to a projection in the usual relational join semantics (specifying which columns will be projected in among Measures and Attributes).

Examples

Given the Data Set DS_1:

DS_1					
Id_1	**Id_2**	**Id_3**	**Me_1**	**Me_2**	**At_1**
2010	A	XX	20	36	E
2010	A	YY	4	9	F
2010	B	XX	9	10	F

Example1: DS_r := DS_1 [keep Me_1] results in:

DS_r			
Id_1	**Id_2**	**Id_3**	**Me_1**
2010	A	XX	20
2010	A	YY	4
2010	B	XX	9

Removal of Components: **drop**

Syntax

op **[drop** comp { **,** comp }* **]**

Input parameters

op the operand
comp a Component to drop

Examples of valid syntaxes

DS_1 [drop Me_2, Me_3]

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input parameters type:

op :: dataset
comp :: name < component >

Result type:

result :: dataset

Additional constraints:

All the Components comp must belong to the input Data Set op.
The Components comp cannot be Identifiers in op.

Behaviour

The operator takes as input a Data Set (op) and some Component names of such a Data Set (comp). These Components can be Measures or Attributes of op but not Identifiers. The operator drops the specified Components and maintains all the other Components of the Data Set. This operation corresponds to a projection in the usual relational join semantics (specifying which columns will be projected out).

Examples

Given the Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Me_1**	**At_1**
2010	A	XX	20	E
2010	A	YY	4	F
2010	B	XX	9	F

Example1: DS_r := DS_1 [drop At_1] results in:

DS_r			
Id_1	**Id_2**	**Id_3**	**Me_1**
2010	A	XX	20
2010	A	YY	4
2010	B	XX	9

Change of Component name : **rename**

Syntax

op **[rename** comp_from **to** comp_to { **,** comp_from **to** comp_to}* **]**

Input Parameters

op the operand
comp_from the original name of the Component to rename
comp_to the new name of the Component after the renaming

Examples of valid syntaxes

DS_1 [rename Me_2 to Me_3]

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input Parameters type

op :: dataset
comp_from :: name < component >
comp_to :: name < component >

Result type

result :: dataset

Additional constraints

The corresponding pairs of Components before and after the renaming (dsc_from and dsc_to) must be defined on the same Value Domain and the same Value Domain Subset.
The components used in dsc_from must belong to the input Data Set and the component used in the dsc_to cannot have the same names as other Components of the result Data Set.

Behaviour

The operator assigns new names to one or more Components (Identifier, Measure or Attribute Components). The resulting Data Set, after renaming the specified Components, must have unique names of all its Components (otherwise a runtime error is raised). Only the Component name is changed and not the Component Values, therefore the new Component must be defined on the same Value Domain and Value Domain Subset as the original Component (see also the IM in the User Manual). If the name of a Component defined on a different

Value Domain or Set is assigned, an error is raised. In other words, **rename** is a transformation of the variable without any change in its values.

Examples

Given the Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Me_1**	**At_1**
1	B	XX	20	F
1	B	YY	1	F
2	A	XX	4	E
2	A	YY	9	F

Example1: DS_r := DS_1 [rename Me_1 to Me_2, At_1 to At_2] results in:

DS_r				
Id_1	**Id_2**	**Id_3**	**Me_2**	**At_2**
1	B	XX	20	F
1	B	YY	1	F
2	A	XX	4	E
2	A	YY	9	F

Pivoting : **pivot**

Syntax

op **[pivot** identifier **,** measure **]**

Input parameters

op the operand
identifier the Identifier Component of op to pivot
measure the Measure Component of op to pivot

Examples of valid syntaxes

DS_1 [pivot Id_2, Me_1]

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input Parameters type

op :: dataset
identifier :: name < identifier >
measure :: name < measure >

Result type

result :: dataset

Additional constraints

The Measures created by the operator according to the behaviour described below must be defined on the same Value Domain as the input Measure.

Behaviour

The operator transposes several Data Points of the operand Data Set into a single Data Point of the resulting Data Set. The semantics of **pivot** can be procedurally described as follows.

1. It creates a virtual Data Set VDS as a copy of op
2. It drops the Identifier Component identifier and all the Measure Components from VDS.
3. It groups VDS by the values of the remaining Identifiers.
4. For each distinct value of identifier in op, it adds a corresponding measure to VDS, named as the value of identifier. These Measures are initialized with the NULL value.
5. For each Data Point of op, it finds the Data Point of VDS having the same values as for the common Identifiers and assigns the value of measure (taken from the current Data Point of op) to the Measure of VDS having the same name as the value of identifier (taken from the Data Point of op).

The result of the last step is the output of the operation.

Note that **pivot** may create Measures whose names are non-regular (i.e. they may contain special characters, reserved keywords, etc.) according to the rules about the artefact names described in the User Manual (see the section "The artefact names" in the chapter "VTL Transformations"). As said in the User Manual, those names must be quoted to be referenced within an expression.

Examples

Given the Data Set DS_1:

DS_1			
Id_1	**Id_2**	**Me_1**	**At_1**
1	A	5	E
1	B	2	F
1	C	7	F
2	A	3	E
2	B	4	E
2	C	9	F

Example1: DS_r := Ds_1 [pivot Id_2, Me_1] results in:

DS_r			
Id_1	**A**	**B**	**C**
1	5	2	7
2	3	4	9

Unpivoting : **unpivot**

Syntax

op **[unpivot** identifier **,** measure **]**

Input parameters

op	the dataset operand
identifier	the Identifier Component to be created
measure	the Measure Component to be created

Examples of valid syntaxes

DS [unpivot Id_5, Me_3]

Semantics for scalar operations

This operator cannot be applied to *scalar* values.

Input Parameters type

op ::	dataset
identifier ::	name < identifier >
measure ::	name < measure >

Result type

result ::	dataset

Additional constraints

All the measures of op must be defined on the same Value Domain.

Behaviour

The **unpivot** operator transposes a single Data Point of the operand Data Set into several Data Points of the result Data set. Its semantics can be procedurally described as follows.

1. It creates a virtual Data Set VDS as a copy of op
2. It adds adds the Identifier Component identifier and the Measure Component measure to VDS.
3. For each Data Point DP and for each Measure M of op whose value is not NULL, the operator inserts a Data Point into VDS whose values are assigned as specified in the following points
4. The VDS Identifiers other than identifier are assigned the same values as the corresponding Identifiers of the op Data Point
5. The VDS identifier is assigned a value equal to the **name** of the Measure M of op
6. The VDS measure is assigned a value equal to the **value** of the Measure M of op

The result of the last step is the output of the operation.

When a Measure is NULL then **unpivot** does not create a Data Point for that Measure.
Note that in general pivoting and unpivoting are not exactly symmetric operations, i.e., in some cases the unpivot operation applied to the pivoted Data Set does not recreate exactly the original Data Set (before pivoting).

Examples

Given the Data Set DS_1:

DS_1			
Id_1	**A**	**B**	**C**
1	5	2	7
2	3	4	9

Example1: DS_r := DS_1 [unpivot Id_2, Me_1] results in:

DS_r		
Id_1	**Id_2**	**Me_1**

1	A	5
1	B	2
1	C	7
2	A	3
2	B	4
2	C	9

Subspace : **sub**

Syntax

op **[sub** identifier **=** value { **,** identifier **=** value }* **]**

Input parameters

op dataset
identifier Identifier Component of the input Data Set op
value valid value for identifier

Examples of valid syntaxes

DS_r := DS_1 [Id_2 = "A", Id_5 = 1]

Semantics for scalar operations

This operator cannot be applied to scalar values.

Input Parameters type

op :: dataset
identifier :: name < identifier >
value :: scalar

Result type

result :: dataset

Additional constraints

The specified Identifier Components identifier(s) must belong to the input Data Set op.
Each Identifier Component can be specified only once.
The specified value must be an allowed value for identifier.

Behaviour

The operator returns a Data Set in a subspace of the one of the input Dataset. Its behaviour can be procedurally described as follows:

1. It creates a virtual Data Set VDS as a copy of op
2. It maintains the Data Points of VDS for which identifier = value (for all the specified identifier) and eliminates all the Data Points for which identifier <> value (even for only one specified identifier)
3. It projects out ("drops", in VTL terms) all the identifier(s)

The result of the last step is the output of the operation.

The resulting Data Set has the Identifier Components that are not specified as identifier(s) and has the same Measure and Attribute Components of the input Data Set.

The result Data Set does not violate the functional constraint because after the filter of the step 2, all the remaining identifier(s) do not contain the same Values for all the Data Points. In other words, given that the input

Data Set is a 1st order function and therefore does not contain duplicates, the result Data Set is a 1st order function as well. To show this, let $K_1,\ldots,K_m,\ldots,K_n$ be the Identifier components for the generic input Data Set DS. Let us suppose that $K_1,\ldots,K_m$ are assigned to fixed values by using the subspace operator. A duplicate could arise only if in the result there are two Data Points DP_{r1} and DP_{r2} having the same value for $K_{m+1},\ldots,K_n$, but this is impossible since such Data Points had same $K_1,\ldots,K_m$ in the original Data Set DS, which did not contain duplicates.

If we consider the vector space of Data Points individuated by the n-uples of Identifier components of a Data Set $DS(K_1,\ldots,K_n,\ldots)$ (along, e.g., with the operators of sum and multiplication), we have that the subspace operator actually performs a subsetting of such space into another space with fewer Identifiers. This can be also seen as the equivalent of a *dice* operation performed on hyper-cubes in multi-dimensional data warehousing.

Examples

Given the Data Set DS_1:

DS_1				
Id_1	**Id_2**	**Id_3**	**Me_1**	**At_1**
1	A	XX	20	F
1	A	YY	1	F
1	B	XX	4	E
1	B	YY	9	F
2	A	XX	7	F
2	A	YY	5	E
2	B	XX	12	F
2	B	YY	15	F

Example1: DS_r := DS_1 [sub Id_1 = 1, Id_2 = "A"] results in:

DS_r		
Id_3	**Me_1**	**At_1**
XX	20	F
YY	1	F

Example 2: DS_r := DS_1 [sub Id_1 = 1, Id_2 = "B", Id_3 = "YY"] results in:

DS_r	
Me_1	**At_1**
9	F

Example 3: DS_r := DS_1 [sub Id_2 = "A"] + DS_1 [sub Id_2 = "B"] results in:

Assuming that At_1 is viral and that in the propagation rule the greater value prevails, results in:

DS_r			
Id_1	**Id_3**	**Me_1**	**At_1**
1	XX	24	F

1	YY	10	F
2	XX	19	F
2	YY	20	F