

SDMX Technical Working Group

VTL Task Force

# **VTL - version 2.0** **(Validation & Transformation Language)**

## **Part 2 - Reference Manual**

*July 2018*

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.

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

82

### 83 **Acknowledgements**

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

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

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

97

# Table of contents

99	<b>Foreword .....</b>	<b>2</b>
100	<b>Table of contents .....</b>	<b>4</b>
101	<b>Introduction .....</b>	<b>8</b>
102	<b>Overview of the language and conventions .....</b>	<b>9</b>
103	Introduction .....	9
104	Conventions for writing VTL Transformations .....	10
105	Typographical conventions .....	11
106	Abbreviations for the names of the artefacts .....	12
107	Conventions for describing the operators' syntax .....	12
108	Description of the data types of operands and result .....	14
109	VTL-ML Operators .....	15
110	VTL-ML - Evaluation order of the Operators .....	27
111	Description of VTL Operators .....	27
112	<b>VTL-DL - Rulesets .....</b>	<b>29</b>
113	define datapoint ruleset .....	29
114	define hierarchical ruleset .....	31
115	<b>VTL-DL – User Defined Operators .....</b>	<b>39</b>
116	define operator .....	39
117	Data type syntax .....	40
118	<b>VTL-ML - Typical behaviours of the ML Operators .....</b>	<b>42</b>
119	Typical behaviour of most ML Operators .....	42
120	Operators applicable on one Scalar Value or Data Set or Data Set Component .....	42
121	Operators applicable on two Scalar Values or Data Sets or Data Set Components .....	43
122	Operators applicable on more than two Scalar Values or Data Set Components .....	45
123	Behaviour of Boolean operators .....	45
124	Behaviour of Set operators .....	46
125	Behaviour of Time operators .....	46
126	Operators changing the data type .....	47
127	Type Conversion and Formatting Mask .....	48
128	The Numbers Formatting Mask .....	48
129	The Time Formatting Mask .....	48
130	Attribute propagation .....	51
131	<b>VTL-ML - General purpose operators .....</b>	<b>53</b>

132	Parentheses : <b>()</b> .....	53
133	Persistent assignment : <b>&lt;-</b> .....	53
134	Non-persistent assignment : <b>:=</b> .....	55
135	Membership : <b>#</b> .....	56
136	User-defined operator call.....	57
137	Evaluation of an external routine : <b>eval</b> .....	58
138	Type conversion : <b>cast</b> .....	59
139	<b>VTL-ML - Join operators</b> .....	<b>64</b>
140	Join : <b>inner_join, left_join, full_join, cross_join</b> .....	64
141	<b>VTL-ML - String operators</b> .....	<b>73</b>
142	String concatenation : <b>  </b> .....	73
143	Whitespace removal : <b>trim, rtrim, ltrim</b> .....	74
144	Character case conversion : <b>upper/lower</b> .....	75
145	Sub-string extraction : <b>substr</b> .....	76
146	String pattern replacement: <b>replace</b> .....	78
147	String pattern location : <b>instr</b> .....	79
148	String length : <b>length</b> .....	81
149	<b>VTL-ML - Numeric operators</b> .....	<b>83</b>
150	Unary plus : <b>+</b> .....	83
151	Unary minus: <b>-</b> .....	84
152	Addition : <b>+</b> .....	85
153	Subtraction : <b>-</b> .....	87
154	Multiplication : <b>*</b> .....	88
155	Division : <b>/</b> .....	90
156	Modulo : <b>mod</b> .....	91
157	Rounding : <b>round</b> .....	93
158	Truncation : <b>trunc</b> .....	95
159	Ceiling : <b>ceil</b> .....	97
160	Floor: <b>floor</b> .....	98
161	Absolute value : <b>abs</b> .....	99
162	Exponential : <b>exp</b> .....	100
163	Natural logarithm : <b>ln</b> .....	101
164	Power : <b>power</b> .....	103
165	Logarithm : <b>log</b> .....	104
166	Square root : <b>sqrt</b> .....	105

167	<b>VTL-ML - Comparison operators.....</b>	<b>107</b>
168	Equal to :     = .....	107
169	Not equal to :     <> .....	108
170	Greater than :     >     >= .....	109
171	Less than :     <     <= .....	111
172	Between : <b>between</b> .....	112
173	Element of: <b>in / not_in</b> .....	114
174	match_characters <b>match_characters</b> .....	116
175	IsNull: <b>isnull</b> .....	117
176	Exists in : <b>exists_in</b> .....	118
177	<b>VTL-ML - Boolean operators.....</b>	<b>121</b>
178	Logical conjunction: <b>and</b> .....	121
179	Logical disjunction : <b>or</b> .....	122
180	Exclusive disjunction : <b>xor</b> .....	124
181	Logical negation : <b>not</b> .....	126
182	<b>VTL-ML - Time operators.....</b>	<b>128</b>
183	Period indicator : <b>period_indicator</b> .....	128
184	Fill time series : <b>fill_time_series</b> .....	129
185	Flow to stock : <b>flow_to_stock</b> .....	135
186	Stock to flow : <b>stock_to_flow</b> .....	138
187	Time shift : <b>timeshift</b> .....	141
188	Time aggregation : <b>time_agg</b> .....	145
189	Actual time : <b>current_date</b> .....	146
190	<b>VTL-ML - Set operators.....</b>	<b>148</b>
191	Union: <b>union</b> .....	148
192	Intersection : <b>intersect</b> .....	149
193	Set difference : <b>setdiff</b> .....	150
194	Simmetric difference : <b>syndiff</b> .....	152
195	<b>VTL-ML - Hierarchical aggregation.....</b>	<b>154</b>
196	Hierarchical roll-up : <b>hierarchy</b> .....	154
197	<b>VTL-ML - Aggregate and Analytic operators.....</b>	<b>158</b>
198	Aggregate invocation .....	159
199	Analytic invocation .....	162
200	Counting the number of data points: <b>count</b> .....	165
201	Minimum value : <b>min</b> .....	166

202	Maximum value : <b>max</b> .....	167
203	Median value : <b>median</b> .....	168
204	Sum : <b>sum</b> .....	169
205	Average value : <b>avg</b> .....	171
206	Population standard deviation : <b>stddev_pop</b> .....	172
207	Sample standard deviation : <b>stddev_samp</b> .....	173
208	Population variance : <b>var_pop</b> .....	174
209	Sample variance : <b>var_samp</b> .....	175
210	First value : <b>first_value</b> .....	176
211	Last value : <b>last_value</b> .....	177
212	Lag : <b>lag</b> .....	179
213	lead : <b>lead</b> .....	180
214	Rank : <b>rank</b> .....	181
215	Ratio to report : <b>ratio_to_report</b> .....	183
216	<b>VTL-ML - Data validation operators</b> .....	<b>185</b>
217	check_datapoint .....	185
218	check_hierarchy .....	187
219	check .....	191
220	<b>VTL-ML - Conditional operators</b> .....	<b>194</b>
221	if-then-else : <b>if</b> .....	194
222	Nvl : <b>nvl</b> .....	196
223	<b>VTL-ML - Clause operators</b> .....	<b>198</b>
224	Filtering Data Points : <b>filter</b> .....	198
225	Calculation of a Component : <b>calc</b> .....	199
226	Aggregation : <b>aggr</b> .....	200
227	Maintaining Components: <b>keep</b> .....	203
228	Removal of Components: <b>drop</b> .....	204
229	Change of Component name : <b>rename</b> .....	205
230	Pivoting : <b>pivot</b> .....	206
231	Unpivoting : <b>unpivot</b> .....	207
232	Subspace : <b>sub</b> .....	209
233		

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

237 The VTL 2.0 library of the Operators is described hereinafter.

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

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

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

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

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

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

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

256



## 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 ;
```

306 All the same the Transformation (ii) would be executed first, because it calculates the Data Set DS\_np which is  
 307 an operand of the Transformation (i).  
 308 When Transformations are written in sequence, a semicolon (;) is used to denote the end of a Transformation  
 309 and the beginning of the following one.  
 310

## 311 Conventions for writing VTL Transformations

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

### 313 Transformations:

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

### 317 Comments:

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

- 319 • A multi-line comment is embedded between */\** and *\*/* and, obviously, can span over several lines:  
 320                   */\** multi-line  
 321                   comment text *\*/*
- 322 • A single-line comment follows the symbol *//* up to the next end of line:  
 323                   *//* text of a comment on a single line
- 324 • A sequence of spaces, tabs, end-of-line characters or comments is considered as a single space.
- 325 • The characters */\**, *\*/*, *//* and the whitespaces can be part of a string literal (within double quotes) but in  
 326 such a case they are part of the string characters and do not have any special meaning.

327  
 328 Examples of valid comments:

329       *Example 1:*

330                   */\** this is a multi-line  
 331                   Comment *\*/*

332       *Example 2:*

333                   *//* this is single-line comment

334       *Example 3:*

335                   DS\_r <- */\** A is a dataset *\*/* A + */\** B is a dataset *\*/* B ;  
 336                   *(for the VTL this statement is the Transformation DS\_r <- A + B; )*

337       *Example 4:*

338                   DS\_r := DS\_1                   *//* my comment  
 339                   \* DS\_2 ;  
 340                   *(for the VTL this statement is the Transformation DS\_r := DS\_1 \* DS\_2; )*

341

## 342

343  
344  
345  
346

347

348

## 349 Abbreviations for the names of the artefacts

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

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

358	<i>Complete name</i>	<i>Compact name</i>	<i>Abbreviation</i>
359	<i>Data Set</i>	<i>Data Set</i>	<i>DS</i>
360	<i>Data Point</i>	<i>Data Point</i>	<i>DP</i>
361	<i>Identifier Component</i>	<i>Identifier</i>	<i>Id</i>
362	<i>Measure Component</i>	<i>Measure</i>	<i>Me</i>
363	<i>Attribute Component</i>	<i>Attribute</i>	<i>At</i>
364	<i>Data Set Component</i>	<i>Component</i>	<i>Comp</i>
365	<i>Value Domain Subset</i>	<i>Subset or Set</i>	<i>Set</i>
366	<i>Value Domain</i>	<i>Domain</i>	<i>VD</i>

367 A positive integer suffix (with or without an underscore) can be added in the end to distinguish more than one  
368 instance of the same artefact (e.g., DS\_1, DS\_2, ..., DS\_N, Me1, Me2, ...MeN ). The suffix “r” stands for the result of  
369 a Transformation (e.g., DS\_r).

## 370 Conventions for describing the operators’ syntax

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

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

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

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

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

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

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

392 **op1 + op2**

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

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

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

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

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

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

403           **substr** ( op, start )

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

405           **substr** ( op )

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

- 410           • non-optional           : *non-optional sub-expression (text without braces)*
- 411           • {optional}            : *optional sub-expression (zero or 1 occurrence)*
- 412           • {non-optional}<sup>1</sup>       : *non-optional sub-expression (just 1 occurrence)*
- 413           • {one-or-more}<sup>+</sup>       : *sub-expression repeatable from 1 to many occurrences*
- 414           • {zero-or-more}<sup>\*</sup>       : *sub-expression repeatable from 0 to many occurrences*
- 415           • { part1 | part2 | part3 }       : *optional alternative sub-expressions (zero or 1 occurrence)*
- 416           • { part1 | part2 | part3 }<sup>1</sup>       : *alternative sub-expressions (just 1 occurrence)*
- 417           • { part1 | part2 | part3 }<sup>+</sup>       : *alternative sub-expressions, from 1 to many occurrences*
- 418           • { part1 | part2 | part3 }<sup>\*</sup>       : *alternative sub-expressions, from 0 to many occurrences*

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

421           sub-expr<sub>1</sub>-text   sub-expr<sub>2</sub>-name   ...   sub-expr<sub>N-1</sub>-name   sub-expr<sub>N</sub>-text

422                           sub-expr<sub>2</sub>-name ::=   sub-expr<sub>2</sub>-text

423                           ... possible others ...

424                           sub-expr<sub>N-1</sub>-name       ::=   sub-expr<sub>N-1</sub>-text

425     In this representation of a meta-expression:

- 426           • *The first line is the text of the meta-expression*
- 427           • sub-expr<sub>1</sub>-text, sub-expr<sub>N</sub>-text   are sub-expressions directly written in the meta-expression
- 428           • sub-expr<sub>2</sub>-name, ... sub-expr<sub>N-1</sub>-name   are identifiers of sub-expressions.
- 429           • sub-expr<sub>2</sub>-text, ... sub-expr<sub>N-1</sub>-text   are subexpression written separately from the meta-expression.
- 430           • The symbol ::= means “is defined as” and denotes the assignment of a sub-expression-text to a sub-  
431           expression-name.

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

434           Meta-expression ::=   { **trim** | **ltrim** | **rtrim** }<sup>1</sup> ( op )

435     The meta-expression above synthesizes that:

- 436           • **trim**, **ltrim**, **rtrim** are the operators’ symbols (reserved keywords);
- 437           • (, )   are symbols of the operators syntax (reserved keywords);
- 438           • op is the only operand of the three operators;
- 439           • “{ }<sup>1</sup>” and “|” are symbols of the meta-syntax; in particular “|” indicates that the three operators are  
440           alternative (a single invocation can contain only one of them) and “{ }<sup>1</sup>” indicates that a single invocation  
441           contains just one of the shown alternatives;

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

443           ltrim ( DS\_2 )

444           rtrim ( DS\_3 )

445     In these invocations, **ltrim** and **rtrim** are the symbols of the invoked operator and DS\_2 and DS\_3 are the names  
446     of the specific Data Sets which are operands respectively of the former and the latter invocation.

447

## 448 Description of the data types of operands and result

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

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

452  
 453 Moreover, the word “name” in the data type description denotes the fact that the argument of the invocation can  
 454 contain only the name of an artefact of such a type but not a sub-expression. For example:

455 comp :: name < component < string > >

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

458 The word “name” added as a suffix to the parameter name means the same (for example if the parameter above  
 459 is called **comp\_name**).

## 460 VTL-ML Operators

461

Name	Symbol	Syntax	Description	Notation	Input parameters type	Result type	Behaviour
Parentheses	<b>()</b>	<b>{ op }</b>	Override the default evaluation order of the operators	Func.	op :: dataset   component   scalar	dataset  component   scalar	Specific
Persistent assignment	<b>&lt;-</b>	re <- op	Assigns an Expression to a persistent model artefact	Infix	re :: name op :: dataset	empty	Specific
Non persistent assignment	<b>:=</b>	re := op	Assigns an Expression to a non persistent model artefact	Infix	re :: name op :: dataset   scalar	empty	Specific
Membership	<b>#</b>	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	<b>eval</b>	<b>eval</b> ( externalRoutineName ( {argument} { , argument }* ) , language, <b>returns</b> outputType )	Evaluates an external routine	Func.	externalRoutineName :: string argument :: any expression language :: string outputType :: outputParameterType	dataset	Specific

Type conversion	cast	cast ( op ,scalarType { , mask } )	converts to the specified data type	Func.	op :: dataset{ measure<scalar> _ }   component<scalar>   scalar  scalarType :: scalar type  mask :: string	dataset{ measure<scalar> _ }   component<scalar>   scalar	Changing data type
Join	<b>inner_join, left_join, full_join, cross_join,</b>	<pre> joinOperator ( ds { as alias } { , ds { as alias } } *     { 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 }<sup>1</sup> calcClause ::= { calcRole } calcComp := calcExpr                { , { calcRole } calcComp := calcExpr } * calcRole ::= { identifier   measure   attribute   viral attribute }<sup>1</sup> aggrClause ::= { aggrRole } aggrComp := aggrExpr                { , { aggrRole } aggrComp := aggrExpr } * aggrRole ::= { measure   attribute   viral attribute }<sup>1</sup> groupingClause ::= { group by idList                       group except idList                       group all conversionExpr }<sup>1</sup>                     { having havingCondition } </pre>	Inner join, left outer join, full outer join, cross join,	Func.	ds :: dataset alias :: 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>	dataset	Specific
String concatenation		op1    op2	Concatenates two strings	Infix	op1, op2 :: dataset { measure<string> _+ }   component<string>   string	dataset { measure<string> _+ }   component<string>   string	On two scalars, DSS or DSCs



Whitespace removal	<b>trim</b> <b>rtrim</b> <b>ltrim</b>	<b>{trim ltrim rtrim}<sup>1</sup> ( op )</b>	Removes trailing or/and leading whitespace from a string	Func.	op :: dataset { measure<string> _+ }   component<string>   string	dataset { measure<string> _+ }   component<string>   string	On one scalar, DS or DSC
Character case conversion	<b>upper</b> <b>lower</b>	<b>{upper   lower}<sup>1</sup> ( op )</b>	Converts the character case of a string in upper or lower case	Func.	op :: dataset { measure<string> _+ }   component<string>   string	dataset { measure<string> _+ }   component<string>   string	On one scalar, DS or DSC
Sub-string extraction	<b>substr</b>	<b>substr ( op, start, length )</b>	Extracts the substring that starts in a specified position and has a specified length	Func.	op :: dataset { measure<string> _+ }   component<string>   string  start :: component < integer[>=1]>   integer[>= 1]  length :: component < integer[>= 0] >   integer[>=0]	dataset { measure<string> _+ }   component<string>   string	On one DS  or  on more than two scalars or DSC
String pattern replacement	<b>replace</b>	<b>replace (op, pattern1, pattern2 )</b>	Replaces a specified string-pattern with another one	Func.	op :: dataset { measure<string> _+ }   component<string>   string  pattern1, pattern2 :: component<string>   string	dataset { measure<string> _+ }   component<string>   string	On one DS  or  on more than two scalars or DSC

String pattern location	<b>instr</b>	<b>instr( op, pattern, start, occurrence )</b>	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	<b>length</b>	<b>length ( op )</b>	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	<b>+</b>	<b>+ op</b>	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	<b>-</b>	<b>- op</b>	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	<b>+</b>	<b>op1 + op2</b>	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	<b>-</b>	<b>op1 - op2</b>	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	<b>*</b>	<b>op1 * op2</b>	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	<b>/</b>	<b>op1 / op2</b>	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	<b>mod</b>	<b>mod ( op1, op2 )</b>	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	<b>round</b>	<b>round ( op, numDigit )</b>	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	<b>trunc</b>	<b>trunc ( op, numDigit )</b>	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	<b>ceil</b>	<b>ceil ( op )</b>	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	<b>floor</b>	<b>floor ( op )</b>	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	<b>abs</b>	<b>abs ( op )</b>	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	<b>exp</b>	<b>exp ( op )</b>	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

Natural logarithm	<b>ln</b>	<b>ln ( op )</b>	Calculates the natural logarithm of a number	Func.	op :: dataset {measure<number[>0]> _+ }   component<number[>0]>   number[>0]	dataset { measure<number> _+ }   component<number>   number	On one scalar, DS or DSC
Power	<b>power</b>	<b>power ( base, exponent)</b>	Raises a number to a certain exponent	Func.	base :: dataset { measure<number> _+ }   component<number>   number  exponent :: component<number>   number	dataset { measure<number> _+ }   component<number>   number	On one DS  or  on two scalars or DSC
Logarithm	<b>log</b>	<b>log ( op, num )</b>	Calculates the logarithm of a number to a certain base	Func.	op :: dataset { measure<number[>1]> _+ }   component<number[>1]>   number[>1]  num:: component<integer[>0]>   integer[>0]	dataset { measure<number> _+ }   component<number>   number	On one DS  or  on two scalars or DSC
Square root	<b>sqrt</b>	<b>sqrt ( op )</b>	Calculates the square root of a number	Func.	op :: dataset { measure<number[>=0]> _+ }   component<number[>= 0]>   number[>= 0]	dataset { measure<number[>=0]> _+ }   component<number[>= 0]>   number[>= 0]	On one scalar, DS or DSC
Equal to	<b>=</b>	left = right	Verifies if two values are equal	Infix	left,right :: dataset {measure<scalar> _ }   component<scalar>   scalar	dataset {measure<boolean> bool_var}   component<boolean>   boolean	Changing data type
Not equal to	<b>&lt;&gt;</b>	left <> right	Verifies if two values are not equal	Infix	left, right :: dataset {measure<scalar> _ }   component<scalar>   scalar	dataset {measure<boolean> bool_var}   component<boolean>   boolean	Changing data type
Greater than	<b>&gt;</b>	left { >   >= } <sup>1</sup> right	Verifies if a first value is greater ( or equal ) than a second value	Infix	left, right :: dataset {measure<scalar> _ }   component<scalar>   scalar	dataset {measure<boolean> bool_var}   component<boolean>   boolean	Changing data type
	<b>&gt;=</b>						
Less than	<b>&lt;</b>	left { <   <= } <sup>1</sup> right	Verifies if a first value is less (or equal) than a second value	Infix	left, right :: dataset {measure<scalar> _ }   component<scalar>   scalar	dataset {measure<boolean> bool_var}   component<boolean>   boolean	Changing data type
	<b>&lt;=</b>						

Between	<b>between</b>	<b>between</b> ( 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	<b>in</b>	op <b>in</b> <u>collection</u> <b>collection</b> ::= 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
	<b>not_in</b>	op <b>not_in</b> <u>collection</u> <u>collection</u> ::= set   valueDomainName	Verifies if a value does not belong to a set of values	Infix			
Match_characters	<b>match_characters</b>	<b>match_characters</b> (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	<b>isnull</b>	<b>isnull</b> ( 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	<b>exists_in</b>	<b>exists_in</b> ( op1, op2, <u>retain</u> ) <u>retain</u> ::= { <b>true</b>   <b>false</b>   <b>all</b> }	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	<b>and</b>	op1 <b>and</b> op2	Calculates the logical AND		op1,op2 :: dataset {measure<boolean> _}   component<boolean>   boolean	dataset { measure<boolean> _}   component<boolean>   boolean	Boolean
Logical disjunction	<b>or</b>	op1 <b>or</b> op2	Calculates the logical OR		op1,op2 :: dataset {measure<boolean> _}   component<boolean>   boolean	dataset { measure<boolean> _}   component<boolean>   boolean	Boolean



Actual time	<b>current_date</b>	<b>current_date ( )</b>	returns the current date	Func.		date	Specific
Union	<b>union</b>	<b>union ( dsList )</b> <u>dsList</u> ::= ds { , ds }*	Computes the union of N datasets	Func.	ds :: dataset	dataset	Set
Intersection	<b>intersect</b>	<b>intersect ( dsList )</b> <u>dsList</u> ::= ds { , ds }*	Computes the intersection of N datasets	Func.	ds :: dataset	dataset	Set
Set difference	<b>setdiff</b>	<b>setdiff ( ds1, ds2 )</b>	Computes the differences of two datasets	Func.	ds1, ds2 :: dataset	dataset	Set
Simmetric difference	<b>symdiff</b>	<b>symdiff ( ds1, ds2 )</b>	Computes the symmetric difference of two datasets	Func.	ds1, ds2 :: dataset	dataset	Set
Hierarchical roll-up	<b>hierarchy</b>	<b>hierarchy ( op , hr { <b>condition</b> condComp { , condComp }* } { <b>rule</b> ruleComp } { <u>mode</u> } { <u>input</u> } { <u>output</u> } )</b> <u>condComp</u> ::= component { , component }* <u>mode</u> ::= <b>non_null</b>   <b>non_zero</b>   <b>partial_null</b>   <b>partial_zero</b>   <b>always_null</b>   <b>always_zero</b> <u>input</u> ::= <b>dataset</b>   <b>rule</b>   <b>rule_priority</b> <u>output</u> ::= <b>computed</b>   <b>all</b>	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		<i>in a Data Set expression:</i> <u>aggregateOperator</u> ( firstOperand { , additionalOperand }* { <u>groupingClause</u> } ) <i>in a Component expression within an aggr clause</i> <u>aggregateOperator</u> ( firstOperand { , additionalOperand }* ) { <u>groupingClause</u> } <u>aggregateOperator</u> ::= <b>avg</b>   <b>count</b>   <b>max</b>   <b>median</b>   <b>min</b>   <b>stddev_pop</b>   <b>stddev_samp</b>   <b>sum</b>   <b>var_pop</b>   <b>var_samp</b> <u>groupingClause</u> ::= { <b>group by</b> groupingId { , groupingId }*   <b>group except</b> groupingId { , groupingId }*   <b>group all</b> conversionExpr }1 { <b>having</b> 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		<p><b>analyticOperator</b>  <b>( firstOperand { , additionalOperand }* <u>over</u> ( <u>analyticClause</u> ) )</b></p> <p><b>analyticOperator ::=</b> <b>avg   count   max   median   min   stddev_pop   stddev_samp   sum   var_pop   var_samp   first_value   lag   last_value   lead   rank   ratio_to_report</b></p> <p><b><u>analyticClause</u> ::=</b>  <b>{ <u>partitionClause</u> } { <u>orderClause</u> } { <u>windowClause</u> }</b></p> <p><b><u>partitionClause</u> ::=</b> <b>partition by</b> identifier { , identifier }*</p> <p><b><u>orderClause</u> ::=</b> <b>order by</b> component { <b>asc</b>   <b>desc</b> } { , component { <b>asc</b>   <b>desc</b> } }*</p> <p><b><u>windowClause</u> ::=</b>  <b>{ data points   range }<sup>1</sup> between <u>limitClause</u> and <u>limitClause</u></b></p> <p><b><u>limitClause</u> ::=</b>  <b>{ num preceding   num following   current data point   unbounded preceding   unbounded following }<sup>1</sup></b></p>	Set of statistical functions used to aggregate data	Func.	<p><b>firstOperand ::</b>  dataset   component</p> <p><b>additionalOperand ::</b> type of the (possible) additional parameter of the invoked operator</p> <p><b>identifier ::</b> name&lt;identifier&gt;</p> <p><b>component ::</b> name&lt;component&gt;</p> <p><b>num ::</b> integer</p>	dataset   component	Specific
Check datapoint	<b>check_datapoint</b>	<p><b>check_datapoint</b>  <b>( op , dpr { components <u>listComp</u> } { <u>output</u> <u>output</u> } )</b></p> <p><b><u>listComp</u> ::=</b> comp { , comp }*</p> <p><b><u>output</u> ::=</b> <b>invalid   all   all_measures</b></p>	Applies one datapoint ruleset on a Data Set	Func.	<p><b>op ::</b> dataset</p> <p><b>dpr ::</b> name &lt; datapoint &gt;</p> <p><b>comp ::</b> name &lt; component &gt;</p>	dataset	Specific
Check hierarchy	<b>check_hierarchy</b>	<p><b>check_hierarchy (</b>  op , hr { <b>condition</b> condComp { , condComp }* }  { <b>rule</b> ruleComp }  { <b>mode</b> } { <b>input</b> } { <b>output</b> } )</p> <p><b><u>mode</u> ::=</b> <b>non_null   non_zero   partial_null   partial_zero   always_null   always_zero</b></p> <p><b><u>input</u> ::=</b> <b>dataset   dataset_priority</b></p> <p><b><u>output</u> ::=</b> <b>invalid   all   all_measures</b></p>	Applies a hierarchical ruleset to a Data Set	Func.	<p><b>op ::</b> dataset</p> <p><b>hr ::</b> name &lt; hierarchical &gt;</p> <p><b>condComp ::</b> name &lt; component &gt;</p> <p><b>ruleComp ::</b> name &lt; identifier &gt;</p>	dataset	Specific
Check	<b>check</b>	<p><b>check ( op { <b>errorcode</b> errorcode } { <b>errorlevel</b> errorlevel } { <b>imbalance</b> imbalance } { <u>output</u> } )</b></p> <p><b><u>output</u> ::=</b> <b>invalid   all</b></p>	Checks if an expression verifies a condition	Func.	<p><b>op ::</b> dataset</p> <p><b>errorcode ::</b> errorcode_vd</p> <p><b>errorlevel ::</b> errorlevel_vd</p> <p><b>imbalance ::</b> number</p>	dataset	Specific



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

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

462

463

## 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.

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

## 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

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

483

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

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

507

509 **define datapoint ruleset**510 *Semantics*

511 The Data Point Ruleset contains Rules to be applied to each individual Data Point of a Data Set for validation  
 512 purposes. These rulesets are also called “horizontal” taking into account the tabular representation of a Data Set  
 513 (considered as a mathematical function), in which each (vertical) column represents a variable and each  
 514 (horizontal) row represents a Data Point: these rulesets are applied on individual Data Points (rows), i.e.,  
 515 horizontally on the tabular representation.

516 *Syntax*

```

517
518
519 define datapoint ruleset rulesetName ( dpRulesetSignature ) is
520   dpRule
521   { ; dpRule }*
522 end datapoint ruleset
523
524   dpRulesetSignature ::= valuedomain listValueDomains | variable listVariables
525   listValueDomains ::= valueDomain { as vdAlias } { , valueDomain { as vdAlias } }*
526   listVariables ::= variable { as varAlias } { , variable { as varAlias } }*
527   dpRule ::= { ruleName : } { when antecedentCondition then } consequentCondition
528               { errorCode errorCode }
529               { errorlevel errorLevel }
530
531 
```

531 *Syntax description*

532	<u>rulesetName</u>	the name of the Data Point Ruleset to be defined.
533	<u>dpRulesetSignature</u>	the Cartesian space of the Ruleset (signature of the Ruleset), which specifies either the
534		Value Domains or the Represented Variables (see the information model) on which the
535		Ruleset is defined. If <b>valuedomain</b> is specified then the Ruleset is applicable to the Data
536		Sets having Components that take values on the specified Value Domains. If <b>variable</b> is
537		specified then the Ruleset is applicable to Data Sets having the specified Variables as
538		Components.
539	<u>valueDomain</u>	a Value Domain on which the Ruleset is defined.
540	<u>vdAlias</u>	an (optional) alias assigned to a Value Domain and valid only within the Ruleset, this can
541		be used for the sake of compactness in writing the Rules. If an alias is not specified then
542		the name of the Value Domain (parameter <u>valueDomain</u> ) is used in the body of the rules.
543	<u>variable</u>	a Represented Variable on which the Ruleset is defined.
544	<u>varAlias</u>	an (optional) alias assigned to a Variable and valid only within the Ruleset, this can be
545		used for the sake of compactness in writing the Rules. If an alias is not specified then the
546		name of the Variable (parameter <u>valueDomain</u> ) is used in the body of the Rules.
547	<u>dpRule</u>	a Data Point Rule, as defined in the following parameters.
548	<u>ruleName</u>	the name assigned to the specific Rule within the Ruleset. If the Ruleset is used for
549		validation then the ruleName identifies the validation results of the various Rules of the
550		Ruleset. The ruleName is optional and, if not specified, is assumed to be the progressive
551		order number of the Rule in the Ruleset. However please note that, if ruleName is
552		omitted, then the Rule names can change in case the Ruleset is modified, e.g., if new Rules
553		are added or existing Rules are deleted, and therefore the users that interpret the
554		validation results must be aware of these changes.
555	<u>antecedentCondition</u>	a <i>boolean</i> expression to be evaluated for each single Data Point of the input Data Set. It
556		can contain Values of the Value Domains or Variables specified in the Ruleset signature
557		and constants; all the VTL-ML component level operators are allowed. If omitted then
558		<u>antecedentCondition</u> is assumed to be TRUE.
559	<u>consequentCondition</u>	a <i>boolean</i> expression to be evaluated for each single Data Point of the input Data Set when
560		the <u>antecedentCondition</u> evaluates to TRUE (as mentioned, missing antecedent

561 conditions are assumed to be TRUE). It contains Values of the Value Domains or Variables  
562 specified in the Ruleset signature and constants; all the VTL-ML component level  
563 operators are allowed. A consequent condition equal to FALSE is considered as a non-  
564 valid result.

565 **errorCode** a literal denoting the error code associated to the rule, to be assigned to the possible non-  
566 valid results in case the Rule is used for validation. If omitted then no error code is  
567 assigned (NULL value). VTL assumes that a Value Domain `errorCode_vd` of error codes  
568 exists in the Information Model and contains all possible error codes: the `errorCode`  
569 literal must be one of the possible Values of such a Value Domain. VTL assumes also that a  
570 Variable `errorCode` for describing the error codes exists in the IM and is a dependent  
571 variable of the Data Sets which contain the results of the validation.

572 **errorLevel** a literal denoting the error level (severity) associated to the rule, to be assigned to the  
573 possible non-valid results in case the Rule is used for validation. If omitted then no error  
574 level is assigned (NULL value). VTL assumes that a Value Domain `errorlevel_vd` of error  
575 levels exists in the Information Model and contains all possible error levels: the  
576 `errorLevel` literal must be one of the possible Values of such a Value Domain. VTL  
577 assumes also that a Variable `errorlevel` for describing the error levels exists in the IM and  
578 is a dependent variable of the Data Sets which contain the results of the validation.

#### 579 *Parameters*

581 **rulesetName** :: name <ruleset >  
582 **valueDomain** :: name < valuedomain >  
583 **vdAlias** :: name  
584 **variable** :: name  
585 **varAlias** :: name  
586 **ruleName** :: name  
587 **antecedentCondition** :: boolean  
588 **consequentCondition** :: boolean  
589 **errorCode** :: `errorCode_vd`  
590 **errorLevel** :: `errorlevel_vd`

#### 592 *Constraints*

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

#### 598 *Semantic specification*

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

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

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

614  
615

---

<sup>1</sup> 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
<b>On Value Domains:</b>  <b>when</b> flow_type = "CREDIT" or flow_type = "DEBIT" <b>then</b> 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.
<b>On Variables:</b>  <b>when</b> flow = "CREDIT" or flow = "DEBIT" <b>then</b> 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.

616

617 The definition of a Ruleset comprises a **signature** (dpRulesetSignature), which specifies the Value Domains or  
618 Variables on which the Ruleset is defined and a set of Rules, that are the Boolean expressions to be applied to  
619 each Data Point. The antecedentCondition and consequentCondition of the Rules can refer only to the Value  
620 Domains or Variables of the Ruleset signature.

621 The Value Domains or the Variables of the Ruleset signature identify the space in which the rules are defined  
622 while each Rule provides for a criterion that demarcates the Set of valid combinations of Values inside this space.  
623 The Data Point Rulesets can be defined in terms of Value Domains in order to maximize their reusability, in fact  
624 this way a Ruleset can be applied on any Data Set which has Components which take values on the Value  
625 Domains of the Ruleset signature. The association between the Components of the Data Set and the Value  
626 Domains of the Ruleset signature is provided by the **check\_datapoint** operator at the invocation of the Ruleset.  
627 When the Ruleset is defined on Variables, their reusability is intentionally limited to the Data Sets which contains  
628 such Variables (and not to other possible Variables which take values from the same Value Domain). If at a later  
629 stage the Ruleset would need to be applied also to other Variables defined on the same Value Domain, a similar  
630 Ruleset should be defined also for the other Variable.

631 Rules are uniquely identified by ruleName. If omitted then ruleName is implicitly assumed to be the progressive  
632 order number of the Rule in the Ruleset. Please note however that, using this default mechanism, the Rule Name  
633 can change if the Ruleset is modified, e.g., if new Rules are added or existing Rules are deleted, and therefore the  
634 users that interpret the validation results must be aware of these changes. In addition, if the results of more than  
635 one Ruleset have to be combined in one Data Set, then the user should make the relevant rulesetNames different.  
636 As said, each Rule is applied in a row-wise fashion to each individual Data Point of a Data Set. The references to  
637 the Value Domains defined in the antecedentCondition and consequentCondition are replaced with the values  
638 of the respective Components of the Data Point under evaluation.

639 .

640

### 641 Examples

642

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

646

```
647 define datapoint ruleset DPR_2 ( variable flow as F, obs_value as O ) is
648     when F = "CREDIT" or F = "DEBIT" then O >= 0 errorcode "Bad value"
649 end datapoint ruleset
```

649

## 650 define hierarchical ruleset

651

### 652 Semantics

653 This operator defines a persistent Hierarchical Ruleset that contains Rules to be applied to individual  
654 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:

```

679         define hierarchical ruleset BeneluxCountriesHierarchy (valuedomain rule Geo_Area ) is
680             BENELUX = BELGIUM + LUXEMBOURG + NETHERLANDS
681         end hierarchical ruleset

```

## Syntax

```

685 define hierarchical ruleset rulesetName ( hrRulesetSignature ) is
686     hrRule
687     { ; hrRule }*
688 end hierarchical ruleset
689
690     hrRulesetSignature ::= vdRulesetSignature | varRulesetSignature
691     vdRulesetSignature ::= valuedomain { condition vdConditioningSignature } rule ruleValueDomain
692     vdConditioningSignature ::= condValueDomain { as vdAlias } { , condValueDomain { as vdAlias } }*
693     varRulesetSignature ::= variable { condition varConditioningSignature } rule ruleVariable
694     varConditioningSignature ::= condVariable { as vdAlias } { , condVariable { as vdAlias } }*
695     hrRule ::= { ruleName : } codeItemRelation { errorcode errorCode } { errorlevel errorLevel }
696     codeItemRelation ::=
697         { when leftCondition then }
698         leftCodeItem { = | > | < | >= | <= }1
699         { + | - } rightCodeItem { [ rightCondition ] }
700         { { + | - }1 rightCodeItem { [ rightCondition ] } }*

```

## Syntax description

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



712		Domain on which the ruleset is defined (i.e., ruleValueDomain) and on the
713		conditioning Value Domains (i.e., condValueDomain).
714	ruleValueDomain	the Value Domain on which the Ruleset is defined
715	condValueDomain	a conditioning Value Domain of the Ruleset
716	vdAlias	an (optional) alias assigned to a Value Domain and valid only within the Ruleset, this
717		can be used for the sake of compactness in writing leftCondition and rightCondition. If
718		an alias is not specified then the name of the Value Domain (i.e., condValueDomain)
719		must be used.
720	<u>varConditioningSignature</u>	the signature of the (possible) conditions of the Ruleset defined on Variables. It
721		specifies the Represented Variables (see the information model) on which these
722		conditions are defined. The Ruleset is meant to be applicable to any Data Set having
723		Components which are defined by the Variable on which the Ruleset is expressed (i.e.,
724		variable) and on the Conditioning Variables.
725	ruleVariable	the variable on which the Ruleset is defined
726	condVariable	a conditioning Variable of the Ruleset
727	varAlias	an (optional) alias assigned to a Variable and valid only within the Ruleset, this can be
728		used for the sake of compactness in writing leftCondition and rightCondition. If an
729		alias is not specified then the name of the Variableomain (parameter condVariable)
730		must be used.
731	ruleName	the name assigned to the specific Rule within the Ruleset. If the Ruleset is used for
732		validation then the ruleName identifies the validation results of the various Rules of
733		the Ruleset. The ruleName is optional and, if not specified, is assumed to be the
734		progressive order number of the Rule in the Ruleset. However please note that, if
735		ruleName is omitted, then the Rule names can change in case the Ruleset is modified,
736		e.g., if new Rules are added or existing Rules are deleted, and therefore the users that
737		interpret the validation results must be aware of these changes. In addition, if the
738		results of more than one Ruleset have to be combined in one Data Set, then the user
739		should make the relevant rulesetNames different.
740	<u>codeItemRelation</u>	specifies a (possibly conditioned) Code Item Relation. It expresses a logical relation
741		between Code Items belonging to the Value Domain of the hrRulesetSignature,
742		possibly conditioned by the Values of the Value Domains or Variables of the
743		Conditioning Signature. The relation is expressed by one of the symbols =, >, >=, <, <=,
744		that in this context denote special logical relationships typical of Code Items. The first
745		member of the relation is a single Code Item. The second member of the relationship
746		is the composition of one or more Code Items combined using the symbols + or -,
747		which in turn also denote special logical operators typical of Code Items. The meaning
748		of these symbols is better explained below and in the User Manual.
749	errorCode	a literal denoting the error code associated to the rule, to be assigned to the possible
750		non-valid results in case the Rule is used for validation. If omitted then no error code
751		is assigned (NULL value). VTL assumes that a Value Domain errorCode_vd of the error
752		codes exists in the Information Model and contains all the possible error codes: the
753		errorCode literal must be one of the possible Values of such a Value Domain. VTL
754		assumes also that a Variable errorCode for describing the error codes exists in the IM
755		and is a dependent variable of the Data Sets which contain the results of the
756		validation.
757	errorLevel	a literal denoting the error level (severity) associated to the rule, to be assigned to the
758		possible non-valid results in case the Rule is used for validation. If omitted then no
759		error level is assigned (NULL value). VTL assumes that a Value Domain errorlevel_vd
760		of the error levels exists in the Information Model and contains all the possible error
761		levels: the errorLevel literal must be one of the possible Values of such a Value
762		Domain. VTL assumes also that a Variable errorlevel for describing the error levels
763		exists in the IM and is a dependent variable of the Data Sets which contain the results
764		of the validation.
765	leftCondition	a <i>boolean</i> expression which defines the pre-condition for evaluating the left member
766		Code Item (i.e., it is evaluated only when the leftCondition is TRUE); It can contain
767		references to the Value domains or the Variables of the conditioningSignature of the
768		Ruleset and Constants; all the VTL-ML component level operators are allowed. The
769		leftCondition is optional, if missing it is assumed to be TRUE and the Rule is always
770		evaluated.
771	leftCodeItem	a Code Item of the Value Domain specified in the hrRulesetSignature.

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

#### 780 *Input parameters type*

781  
 782  
 783 rulesetName :: name < ruleset >  
 784 ruleValueDomain :: name <valuedomain >  
 785 condValueDomain :: name <valuedomain >  
 786 vdAlias :: name  
 787 ruleVariable :: name  
 788 condVariable :: name  
 789 varAlias :: name  
 790 ruleName :: name  
 791 errorCode :: errorcode\_vd  
 792 errorLevel :: errorlevel\_vd  
 793 leftCondition :: boolean  
 794 leftCodeItem :: name  
 795 rightCodeItem :: name  
 796 rightCondition :: boolean

#### 797 *Constraints*

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

#### 803 *Semantic specification*

804 This operator defines a Hierarchical Ruleset named rulesetName that can be used both for validation and  
 805 calculation purposes (see **check\_hierarchy** and **hierarchy**). A Hierarchical Ruleset is a set of Rules expressing  
 806 logical relationships between the Values (Code Items) of a Value Domain or a Represented Variable.

807 Each rule contains a Code Item Relation, possibly conditioned, which expresses the **relation between Code**  
 808 **Items** to be enforced. In the relation, the left member Code Item is put in relation to a combination of one or  
 809 more right member Code Items. The kinds of relations are described below.

810 The left member Code Item can be optionally conditioned through a leftCondition, a *boolean* expression which  
 811 defines the cases in which the Rule has to be applied (if not declared the Rule is applied ever). The participation  
 812 of each right member Code Item in the Relation can be optionally conditioned through a rightCondition, a  
 813 *boolean* expression which defines the cases in which the Code Item participates in the relation (if not declared  
 814 the Code Item participates to the relation ever).

815 As for the mathematical meaning of the relation, please note that each Value (Code Item) is the representation of  
 816 an event belonging to a space of events (i.e., the relevant Value Domain), according to the notions of “event” and  
 817 “space of events” of the probability theory (see also the section on the Generic Models for Variables and Value  
 818 Domains in the VTL IM). Therefore the relations between Values (Code Items) express logical implications  
 819 between events.

820 The envisaged types of relations are: “coincides” (=), “implies” (<), “implies or coincides” (<=), “is implied by”  
 821 (>), “is implied by or coincides” (>=)<sup>2</sup>. For example:

822 *UnitedKingdom < Europe*

823 means that UnitedKingdom implies Europe (if a point belongs to United Kingdom it also belongs to Europe).

824 *January2000 < year2000*

825 means that January of the year 2000 implies the year 2000 (if a time instant belongs to “January 2000” it also  
 826 belongs to the “year 2000”)

827 The first member of a Relation is a single Code Item. The second member can be either a single Code Item, like in  
 828 the example above, or a **logical composition of Code Items** giving another Code Item as result. The logical

---

<sup>2</sup> “Coincides” means “implies and is implied”

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
```

```

948         ; America = Argentina + ... + Venezuela
949         ; Asia = Afghanistan + ... + Yemen
950         ; Europe = Albania + ... + VaticanCity
951         ; Oceania = Australia + ... + Vanuatu
952         ; Afghanistan = AF_reg_01 + ... + AF_reg_N
953         ... ..
954         ; Zimbabwe = ZW_reg_01 + ... + ZW_reg_M
955         ; EuropeanUnion = ... + ... + ... + ...
956         ; CentralAmericaCommonMarket = ... + ... + ... + ...
957         ; OECD_Area = ... + ... + ... + ...
958     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.

```

965     define hierarchical ruleset TransportBreakdown ( variable rule BoPItem ) is
966         transport_method1 : Transport = AirTransport + SeaTransport + LandTransport
967         ; transport_method2 : Transport = PassengersTransport + FreightsTransport
968     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:

```

974     define hierarchical ruleset CalcCountryLevel ( valuedomain condition Residence rule GeoArea) is
975         when Residence = "resident" then Country1 = Country1
976         ; when Residence = "non-resident" then Country1 = Region11 + ... + Region1M
977         ...
978         ; when Residence = "resident" then CountryN = CountryN
979         ; when Residence = "non-resident" then CountryN = Region N1 + ... + RegionNM
980     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.

```

999     define hierarchical ruleset sex_hr (valuedomain rule sex) is
1000         TOTAL = MALE + FEMALE
1001     end hierarchical ruleset

```

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

```

1006     define hierarchical ruleset BeneluxCountriesHierarchy (valuedomain rule GeoArea) is
1007         BENELUX = BELGIUM + LUXEMBOURG + NETHERLANDS errorcode "Bad value for Benelux"

```

1008       end hierarchical ruleset

1009

1010 3) American economic partners. The first rule states that the value for North America should be greater than the  
1011 value reported for US. This type of validation is useful when the data communicated by the data provider do not  
1012 cover the whole composition of the aggregate but only some elements. No conditions are defined.

1013

1014       define hierarchical ruleset american\_partners\_hr (variable rule PartnerArea) is

1015           NORTH\_AMERICA > US

1016           ; SOUTH\_AMERICA = BR + UY + AR + CL

1017       end hierarchical ruleset

1018

1019 4) Example of an aggregate Code Item having multiple definitions to be used for validation only. The Balance of  
1020 Payments item "Transport" can be broken down by type of carrier (Air transport, Sea transport, Land transport)  
1021 and by type of objects transported (Passengers and Freights) and both breakdowns must sum up to the total  
1022 "Transport" figure.

1023

1024       define hierarchical ruleset validationruleset\_bop (variable rule BoPItem ) is

1025           transport\_method1 : Transport = AirTransport + SeaTransport + LandTransport

1026           ; transport\_method2 : Transport = PassengersTransport + FreightsTransport

1027       end hierarchical ruleset

1028

1029

1031

define operator

1032

*Syntax*

1033

**define operator** operator\_name ( { parameter { , parameter }\* } )

1034

{ **returns** outputType }

1035

**is** operatorBody

1036

**end operator**

1037

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

1038

1039

1040

*Syntax description*

1041

operator\_name

the name of the operator

1042

parameter

the names of parameters, their data types and defaultvalues

1043

outputType

the data type of the artefact returned by the operator

1044

operatorBody

the expression which defines the operation

1045

parameterName

the name of the parameter

1046

parameterType

the data type of the parameter

1047

parameterDefaultValue

the default value for the parameter (optional).

1048

1049

*Parameters*

1050

operator\_name

name

1051

outputType

a VTL data type as defined in outputParameterType (see the Data Type Syntax)

1052

operatorBody

a VTL expression having the parameters (i.e., parameterName) as the operands

1053

parameterName

name

1054

parameterType

a VTL data type as defined in inputParameterType (see the Data Type Syntax)

1055

parameterDefaultValue

a Value of the same type as the parameter

1056

1057

1057

*Constraints*

1058

- Each parameterName must be unique within the list of parameters

1059

- parameterDefaultValue must be of the same data type as the corresponding parameter

1060

- if outputType is specified then the type of operatorBody must be compatible with outputType

1061

- If outputType is omitted then the type returned by the operatorBody expression is assumed

1062

- If parameterDefaultValue is specified then the parameter is optional

1063

1064

*Semantic specification*

1065

This operator defines a user-defined Operator by means of a VTL expression, specifying also the parameters,

1066

their data types, whether they are mandatory or optional and their (possible) default values.

1067

1068

*Examples*

1069

*Example1:*

1070

define operator max1 (x integer, y integer)

1071

returns boolean is

1072

if x > y then x else y

1073

end operator

1074

1075

*Example2:*

1076

define operator add (x integer default 0, y integer default 0)

1077

returns number is

1078

x+y

1079

end 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 | scalarSetType | componentType | datasetType | operatorType | rulesetType

scalarType ::= { basicScalarType | valueDomainName | setName }<sup>1</sup> { scalarTypeConstraint } { { **not** } **null** }

basicScalarType ::= **scalar** | **number** | **integer** | **string** | **boolean** | **time** | **date** | **time\_period** | **duration**

scalarTypeConstraint ::= [ valueBooleanCondition ] | { scalarLiteral { , scalarLiteral }<sup>\*</sup> }

scalarSetType ::= **set** { < scalarType > }

componentType ::= componentRole { < scalarType > }

componentRole ::= **component** | **identifier** | **measure** | **attribute** | **viral attribute**

datasetType ::= **dataset** { { componentConstraint { , componentConstraint }<sup>\*</sup> } }

componentConstraint ::= componentType { componentName | multiplicityModifier }<sup>1</sup>

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

operatorType ::= inputParameterType { \* inputParameterType }<sup>\*</sup> -> outputParameterType

inputParameterType ::= scalarType | scalarSetType | componentType | datasetType | rulesetType

outputParameterType ::= scalarType | componentType | datasetType

rulesetType ::= **ruleset** | dpRuleset | hrRuleset

dpRuleset ::= **datapoint**

| **datapoint\_on\_valuedomains** { { valueDomainName { \* valueDomainName }<sup>\*</sup> } }

| **datapoint\_on\_variables** { { variableName { \* variableName }<sup>\*</sup> } }

hrRuleset ::= **hierarchical**

| **hierarchical\_on\_valuedomains** { { valueDomainName

{ ( condValueDomainName { \* condValueDomainName }<sup>\*</sup> ) } }

| **hierarchical\_on\_variables** { { variableName

{ ( condVariableName { \* condVariableName }<sup>\*</sup> ) } }

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:



1117	integer { 0, 1 }	means an integer number whose value is 0 or 1
1118	number [ value >= 0 ]	means a number greater or equal than 0
1119	string { "A", "B", "C" }	means a string whose value is A, B or C:
1120	string [ length (value) <= 10 ]	means a string whose length is lower or equal than 10:
1121		

1122 General examples of the syntax for defining types can be found in the User Manual, section VTL Data Types and  
1123 in the declaration of the data types of the VTL operators (sub-sections “input parameters type” and “result  
1124 type”).

1125

## VTL-ML - Typical behaviours of the ML Operators

1126  
1127  
1128

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.

1129

### Typical behaviour of most ML Operators

1130  
1131

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

1132  
1133  
1134

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.

1135  
1136  
1137

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.

1138  
1139  
1140

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.

1141  
1142

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

1143  
1144

#### *Operations on Scalar values*

1145  
1146

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

1147

#### *Operations on Data Sets*

1148  
1149

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

1149

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

1150

**DS\_r := f( DS\_1 )**

1151

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

1152

**DS\_r := op DS\_1**

1153  
1154

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.

1155  
1156  
1157  
1158  
1159

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.

1160  
1161

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.

1162  
1163

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.

1164

For each Data Point DP\_1 and for each viral Attribute, the value of the Attribute propagates unchanged in DP\_r.

1165  
1166  
1167  
1168

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).

1169

## 1170 *Operations on Data Set Components*

1171 The operator is applied on a Component (COMP\_1) of a Data Set (DS\_1) and returns another Component  
1172 (COMP\_r) which alters the structure of DS\_1 in order to produce the result Data Set (DS\_r).

1173 For example, using a functional style and denoting the operator with  $f( \dots )$ , this can be written as:

1174  $DS_r := DS_1 [ \text{calc } COMP_r := f( COMP_1 ) ]$

1175 The same operation, using an infix style and denoting the operator as **op**, can be written as:

1176  $DS_r := DS_1 [ \text{calc } COMP_r := \text{op } COMP_1 ]$

1177 This means that the operator is applied on COMP\_1 in order to calculate COMP\_r.

- 1178 • If COMP\_r is a new Component which originally did not exist in DS\_1, it is added to the original Components  
1179 of DS\_1, by default as a Measure (unless otherwise specified), in order to produce DS\_r.
- 1180 • If COMP\_r is one of the original Measures or Attributes of DS\_1, the values obtained from the application of  
1181 the operator  $f( \dots )$  replace the DS\_1 original values for such a Measure or Attribute in order to produce  
1182 DS\_r.
- 1183 • If COMP\_r is one of the original Identifiers of DS\_1, the operation is not allowed, because the result can  
1184 become inconsistent.

1185 In any case, an operation on the Components of a Data Set produces a new Data Set, as in the example above.

1186 The application of the operator is allowed only if the input Component belongs to a data type compatible with  
1187 the operator (for example, a numeric operator is applicable only on numeric Components). As already said,  
1188 COMP\_r cannot have the same name of an Identifier of DS\_1.

1189 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  
1190 returning the value of COMP\_r.

1191 As for the data structure, like for the operations on Data Sets above, the result Data Set (DS\_r) has the Identifiers  
1192 and the Measures of the operand Data Set (DS\_1), and has the Attributes resulting from the application of the  
1193 attribute propagation rules on the Attributes of the operand Data Set (DS\_r maintains the Attributes declared as  
1194 “viral” in DS\_1; these Attributes are considered as “viral” also in DS\_r, the “non-viral” Attributes of DS\_1 are not  
1195 kept in DS\_r). If an Attribute is explicitly calculated, the attribute propagation rule is overridden.

1196 Moreover, in the case of the operations on Data Set Components, the (possible) new Component DS\_r can be  
1197 added to the original structure, the role of a (possible) existing DS\_1 Component can be altered, the virality of a  
1198 (possibly) existing DS\_r Attribute can be altered, a (possible) COMP\_r non-viral Attribute can be kept in the  
1199 result. For the alteration of role and virality see also the **calc** clause.

## 1200 Operators applicable on two Scalar Values or Data Sets or Data Set 1201 Components

### 1202 *Operation on Scalar values*

1204 The operator is applied on two Scalar values and returns a Scalar value.

### 1205 *Operation on Data Sets*

1207 The operator is applied either on two Data Sets or on one Data Set and one Scalar value and returns a Data Set.  
1208 The composition of a Data Set and a Component is not allowed (it makes no sense).

1209 For example, using a functional style and denoting the operator with  $f( \dots )$ , this can be written as:

1210  $DS_r := f( DS_1, DS_2 )$

1211 The same kind of operation, using an infix style and denoting the operator as **op**, can be also written as

1212  $DS_r := DS_1 \text{ op } DS_2$

1213 This means that the operator is applied to the values of all the couples of Measures of DS\_1 and DS\_2 having the  
1214 same names in order to produce homonymous Measures in DS\_r. DS\_1 or DS\_2 may be replaced by a Scalar  
1215 value.

1216 The composition of two Data Sets (DS\_1, DS\_2) is allowed if the two operand Data Sets have exactly the same  
1217 Measures and if all these Measures belong to a data type compatible with the operator (for example, a numeric  
1218 operator is applicable only if all the Measures of the operand Data Sets are numeric). If the Measures of the  
1219 operand Data Sets are different or of different types not all compatible with the operator to be applied, the  
1220 membership or the **keep** clauses can be used to select only the proper Measures. The composition is allowed if

1221 these operand Data Sets have the same Identifiers or if one of them has at least all the Identifiers of the other one  
1222 (in other words, the Identifiers of one of the Data Sets must be a superset of the Identifiers of the other one). No  
1223 applicability constraints exist on the Attributes, which can be any.

1224 As for the data content, the operand Data Sets (DS<sub>1</sub>, DS<sub>2</sub>) are joined to find the couples of Data Points (DP<sub>1</sub>,  
1225 DP<sub>2</sub>), where DP<sub>1</sub> is from the first operand (DS<sub>1</sub>) and DP<sub>2</sub> from the second operand (DS<sub>2</sub>), which have the  
1226 same values as for the common Identifiers. Data Points that are not coupled are left out (the inner join is used).  
1227 An operand Scalar value is treated as a Data Point that couples with all the Data Points of the other operand. For  
1228 each couple (DP<sub>1</sub>, DP<sub>2</sub>) a result Data Point (DP<sub>r</sub>) is returned, having for the Identifiers the same values as  
1229 DP<sub>1</sub> and DP<sub>2</sub>.

1230 For each Measure and for each couple (DP<sub>1</sub>, DP<sub>2</sub>), the Measure values of DP<sub>1</sub> and DP<sub>2</sub> are composed through  
1231 the operator so returning the Measure value of DP<sub>r</sub>. An operand Scalar value is composed with all the Measures  
1232 of the other operand.

1233 For each couple (DP<sub>1</sub>, DP<sub>2</sub>) and for each Attribute that propagates in DP<sub>r</sub>, the Attribute value is calculated by  
1234 applying the proper Attribute propagation algorithm on the values of the Attributes of DP<sub>1</sub> and DP<sub>2</sub>.

1235 As for the data structure, the result Data Set (DS<sub>r</sub>) has all the Identifiers (with no repetition of common  
1236 Identifiers) and the Measures of both the operand Data Sets, and has the Attributes resulting from the  
1237 application of the attribute propagation rules on the Attributes of the operands (DS<sub>r</sub> maintains the Attributes  
1238 declared as “viral” for the operand Data Sets; these Attributes are considered as “viral” also in DS<sub>r</sub>, the “non-  
1239 viral” Attributes of the operand Data Sets are not kept in DS<sub>r</sub>).

1240

#### 1241 *Operation on Data Set Components*

1242 The operator is applied either on two Data Set Components (COMP<sub>1</sub>, COMP<sub>2</sub>) belonging to the same Data Set  
1243 (DS<sub>1</sub>) or on a Component and a Scalar value, and returns another Component (COMP<sub>r</sub>) which alters the  
1244 structure of DS<sub>1</sub> in order to produce the result Data Set (DS<sub>r</sub>). The composition of a Data Set and a Component  
1245 is not allowed (it makes no sense).

1246 For example, using a functional style and denoting the operator with **f**( ... ), this can be written as:

1247 
$$DS_r := DS_1 [ \text{calc } COMP_r := f( COMP_1, COMP_2 ) ]$$

1248 The same operation, using an infix style and denoting the operator as **op**, can be written as:

1249 
$$DS_r := DS_1 [ \text{calc } COMP_r := COMP_1 \text{ op } COMP_2 ]$$

1250 This means that the operator is applied on COMP<sub>1</sub> and COMP<sub>2</sub> in order to calculate COMP<sub>r</sub>.

- 1251 • If COMP<sub>r</sub> is a new Component which originally did not exist in DS<sub>1</sub>, it is added to the original Components  
1252 of DS<sub>1</sub>, by default as a Measure (unless otherwise specified), in order to produce DS<sub>r</sub>.
- 1253 • If COMP<sub>r</sub> is one of the original Measures or Attributes of DS<sub>1</sub>, the values obtained from the application of  
1254 the operator **f**( ... ) replace the DS<sub>1</sub> original values for such a Measure or Attribute in order to produce  
1255 DS<sub>r</sub>.
- 1256 • If COMP<sub>r</sub> is one of the original Identifiers of DS<sub>1</sub>, the operation is not allowed, because the result can  
1257 become inconsistent.

1258 In any case, an operation on the Components of a Data Set produces a new Data Set, like in the example above.

1259 The composition of two Data Set Components is allowed provided that they belong to the same Data Set<sup>3</sup>.  
1260 Moreover, the input Components must belong to data types compatible with the operator (for example, a  
1261 numeric operator is applicable only on numeric Components). As already said, COMP<sub>r</sub> cannot have the same  
1262 name of an Identifier of DS<sub>1</sub>.

1263 As for the data content, for each Data Point of DS<sub>1</sub>, the values of COMP<sub>1</sub> and COMP<sub>2</sub> are composed through  
1264 the operator so returning the value of COMP<sub>r</sub>.

1265 As for the data structure, the result Data Set (DS<sub>r</sub>) has the Identifiers and the Measures of the operand Data Set  
1266 (DS<sub>1</sub>), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes  
1267 of the operand Data Set (DS<sub>r</sub> maintains the Attributes declared as “viral” in DS<sub>1</sub>; these Attributes are  
1268 considered as “viral” also in DS<sub>r</sub>, the “non-viral” Attributes of DS<sub>1</sub> are not kept in DS<sub>r</sub>). If an Attribute is  
1269 explicitly calculated, the attribute propagation rule is overridden.

1270 Moreover, in the case of the operations on Data Set Components, a (possible) new Component DS<sub>r</sub> can be added  
1271 to the original structure of DS<sub>1</sub>, the role of a (possibly) existing DS<sub>1</sub> Component can be altered, the virality of a

---

<sup>3</sup> As obvious, the input Data Set can be the result of a previous composition of more other Data Sets, even within the same expression

1272 (possibly) existing DS\_r Attributes can be altered, a (possible) COMP\_r non-viral Attribute can be kept in the  
1273 result. For the alteration of role and virality see also the **calc** clause.

## 1274 Operators applicable on more than two Scalar Values or Data Set 1275 Components

1276 The cases in which an operator can be applied on more than two Data Sets (like the Join operators) are described  
1277 in the relevant sections.

### 1278 *Operation on Scalar values* 1279

1280 The operator is applied on more Scalar values and returns a Scalar value according to its semantics.

1281

### 1282 *Operation on Data Set Components*

1283 The operator is applied either on a combination of more than two Data Set Components (COMP\_1, COMP\_2)  
1284 belonging to the same Data Set (DS\_1) or Scalar values, and returns another Component (COMP\_r) which alters  
1285 the structure of DS\_1 in order to produce the result Data Set (DS\_r). The composition of a Data Set and a  
1286 Component is not allowed (it makes no sense).

1287 For example, using a functional style and denoting the operator with **f**( ... ), this can be written as:

1288 **DS\_r := DS\_1 [ substr COMP\_r := f ( COMP\_1, COMP\_2, COMP\_3 ) ]**

1289 This means that the operator is applied on COMP\_1, COMP\_2 and COMP\_3 in order to calculate COMP\_r.

- 1290 • If COMP\_r is a new Component which originally did not exist in DS\_1, it is added to the original Components  
1291 of DS\_1, by default as a Measure (unless otherwise specified), in order to produce DS\_r.
- 1292 • If COMP\_r is one of the original Measures or Attributes of DS\_1, the values obtained from the application of  
1293 the operator **f**( ... ) replace the DS\_1 original values for such a Measure or Attribute in order to produce  
1294 DS\_r.
- 1295 • If COMP\_r is one of the original Identifiers of DS\_1, the operation is not allowed, because the result can  
1296 become inconsistent.

1297 In any case, an operation on the Components of a Data Set produces a new Data Set, like in the example above.

1298 The composition of more Data Set Components is allowed provided that they belong to the same Data Set<sup>4</sup>.  
1299 Moreover, the input Components must belong to data types compatible with the operator (for example, a  
1300 numeric operator is applicable only on numeric Components). As already said, COMP\_r cannot have the same  
1301 name of an Identifier of DS\_1.

1302 As for the data content, for each Data Point of DS\_1, the values of COMP\_1, COMP\_2 and COMP\_3 are composed  
1303 through the operator so returning the value of COMP\_r.

1304 As for the data structure, the result Data Set (DS\_r) has the Identifiers and the Measures of the operand Data Set  
1305 (DS\_1), and has the Attributes resulting from the application of the attribute propagation rules on the Attributes  
1306 of the operand Data Set (DS\_r maintains the Attributes declared as “viral” in DS\_1; these Attributes are  
1307 considered as “viral” also in DS\_r, the “non-viral” Attributes of DS\_1 are not kept in DS\_r). If an Attribute is  
1308 explicitly calculated, the attribute propagation rule is overridden.

1309 Moreover, in the case of the operations on Data Set Components, a (possible) new Component DS\_r can be added  
1310 to the original structure of DS\_1, the role of a (possibly) existing DS\_1 Component can be altered, the virality of a  
1311 (possibly) existing DS\_r Attributes can be altered, a (possible) COMP\_r non-viral Attribute can be kept in the  
1312 result. For the alteration of role and virality see also the **calc** clause.

1313

## 1314 Behaviour of Boolean operators

1315 The Boolean operators are allowed only on operand Data Sets that have a single measure of type *boolean*. As for  
1316 the other aspects, the behaviour is the same as the operators applicable on one or two Data Sets described above.

---

<sup>4</sup> As obvious, the input Data Set can be the result of a previous composition of more other Data Sets, even within the same expression

## 1317 Behaviour of Set operators

1318 These operators apply the classical set operations (union, intersection, difference, symmetric differences) to the  
 1319 Data Sets, considering them as sets of Data Points. These operations are possible only if the Data Sets to be  
 1320 operated have the same data structure, and therefore the same Identifiers, Measures and Attributes<sup>5</sup>.

## 1321 Behaviour of Time operators

1322 The *time* operators are the operators dealing with *time*, *date* and *time\_period* basic scalar types. These types are  
 1323 described in the User Manual in the sections “Basic Scalar Types” and “External representations and literals used  
 1324 in the VTL Manuals”.

1325 The time-related formats used for explaining the time operators are the following (they are described also in the  
 1326 User Manual).

1327 For the *time* values:

1328 *YYYY-MM-DD/YYYY-MM-DD*

1329 Where *YYYY* are 4 digits for the year, *MM* two digits for the month, *DD* two digits for the day. For  
 1330 example:

1331 2000-01-01/2000-12-31 the whole year 2000

1332 2000-01-01/2009-12-31 the first decade of the XXI century

1333 For the *date* values:

1334 *YYYY-MM-DD*

1335 The meaning of the symbols is the same as above. For example:

1336 2000-12-31 the 31<sup>st</sup> December of the year 2000

1337 2010-01-01 the first of January of the year 2010

1338 For the *time\_period* values:

1339 *YYYY{P}{NNN}*

1340 Where *YYYY* are 4 digits for the year, *P* is one character for the period indicator of the regular period (it  
 1341 refers to the *duration* data type and can assume one of the possible values listed below), *NNN* are from  
 1342 zero to three digits which contain the progressive number of the period in the year. For annual data the  
 1343 A and the three digits *NNN* can be omitted. For example:

1344 2000M12 the month of December of the year 2000 (duration: M)

1345 2010Q1 the first quarter of the year 2010 (duration: Q)

1346 2010A the whole year 2010 (duration: A)

1347 2010 the whole year 2010 (duration: A)

1348 For the *duration* values, which are the possible values of the period indicator of the regular periods above, it is  
 1349 used for simplicity just one character whose possible values are the following:

1350	<u>Code</u>	<u>Duration</u>
1351	D	Day
1352	W	Week
1353	M	Month
1354	Q	Quarter
1355	S	Semester
1356	A	Year

1357 As mentioned in the User Manual, these are only examples of possible time-related representations, each VTL  
 1358 system is free of adopting different ones. In fact no predefined representations are prescribed, VTL systems are  
 1359 free to using they preferred or already existing ones.

1360 Several time operators deal with the specific case of Data Sets of time series, having an Identifier component that  
 1361 acts as the reference time and can be of one of the scalar types *time*, *date* or *time\_period*; moreover this Identifier  
 1362 must be periodical, i.e. its possible values are regularly spaced and therefore have constant duration (frequency).

---

<sup>5</sup> According to the VTL IM, the Variables that have the same name have also the same data type

1363 It is worthwhile to recall here that, in the case of Data Sets of time series, VTL assumes that the information  
1364 about which is the Identifier Components that acts as the reference time and which is the period (frequency) of  
1365 the time series exists and is available in some way in the VTL system. The VTL Operators are aware of which is  
1366 the reference time and the period (frequency) of the time series and use these information to perform correct  
1367 operations. VTL also assumes that a Value Domain representing the possible periods (e.g. the period indicator  
1368 Value Domain shown above) exists and refers to the *duration* scalar type. For the assumptions above, the users  
1369 do not need to specify which is the Identifier Component having the role of reference time.

1370 The operators for time series can be applied only on Data Sets of time series and returns a Data Set of time  
1371 series. The result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set  
1372 and contains the same time series as the operand. The Attribute propagation rule is not applied.

1373 

## Operators changing the data type

1374 These Operators change the Scalar data type of the operands they are applied to (i.e. the type of the result is  
1375 different from the type of the operand). For example, the **length** operator is applied to a value of *string* type and  
1376 returns a value of *integer* type. Another example is the **cast** operator.

1377 *Operation on Scalar values*

1378 The operator is applied on (one or more) Scalar values and returns one Scalar value of a different data type.

1380 *Operation on Data Sets*

1381 If an Operator change the data type of the Variable it is applied to (e.g., from *string* to *number*), the result Data Set  
1382 cannot maintain this Variable as it happens in the previous cases, because a Variable cannot have different data  
1383 types in different Data Sets<sup>6</sup>.

1384 As a consequence, the converted variable cannot follow the same rules described in the sections above and must  
1385 be replaced, in the result Data Set, by another Variable of the proper data type.

1386 For sake of simplicity, the operators changing the data type are allowed only on mono-measure operand Data  
1387 Sets, so that the conversion happens on just one Measure. A default generic Measure is assigned by default to the  
1388 result Data Set, depending on the data type of the result (the default Measure Variables are reported in the table  
1389 below).

1390 Therefore, if the operands are originally multi-measure, just one Measure must be pre-emptively selected (for  
1391 example through the membership operator) in order to apply the changing-type operator. Moreover, if in the  
1392 result Data Set a different Measure Variable name is desired than the one assigned by default, it is possible to  
1393 change the Variable name (see the **rename** operator).

1394 As for the Identifiers and the Attributes, the behaviour of these operators is the same as the typical behaviour of  
1395 the unary or binary operators.

1397 *Operation on Data Set Components*

1398 For the same reasons above, the result Component cannot be the same as one of the operand Components and  
1399 must be of the appropriate Scalar data type.

1401 *Default Names for Variables and Value Domains used in this manual*

1402 The following table shows the default Variable names and the relevant default Value Domain. These are only the  
1403 names used in this manual for explanatory purposes and can be personalised in the implementations. If VTL  
1404 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

<sup>6</sup> 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 5<sup>th</sup>)
- 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”)



- 1442 - The number of letters matches the number of digits, for example YYYY means that the year is represented  
1443 with four digits and MM that the month is of 2 digits
- 1444 - The numerical representation is assumed to be padded by leading 0 by default, for example MM means that  
1445 April is represented as 04 and the year 33 AD as 0033
- 1446 - If the numerical representation is not padded, the optional digits that can be omitted (if equal to zero) are  
1447 enclosed within braces; for example {M}M means that April is represented by 4 and December by 12, while  
1448 {YYY}Y means that the 33 AD is represented by 33

1449 For textual representations of the time units:

- 1450 - **Special words** denote a textual localized representation of a certain unit, for example DAY means a textual  
1451 representation of the day (MONDAY, TUESDAY ...)
- 1452 - An optional number following the special word denote the maximum length, for example DAY3 is a textual  
1453 representation that uses three characters (MON, TUE ...)
- 1454 - The case of the special word correspond to the case of the value; for example day3 (lowercase) denotes the  
1455 values mon, tue ...
- 1456 - The case of the initial character of the special word correspond to the case of the initial character of the time  
1457 format; for example Day3 denotes the values Mon, Tue ...
- 1458 - The letter P denotes the period indicator, (i.e., day, week, month ...) and the letter p denotes one digit for the  
1459 number of periods

1460 Representation of more time units:

- 1461 - If more time units are used in the same mask (for example years, months, days), it is assumed that the more  
1462 detailed units (e.g., the day) are expressed through the order number that they assume within the less  
1463 detailed ones (e.g., the month and the year). For example, if years, weeks and days are used, the weeks are  
1464 within the year (from 1 to 53) and the days are within the year and the week (from 1 to 7).
- 1465 - The position of the digits in the mask denotes the position of the corresponding values; for example,  
1466 YYYYMMDD means four digits for the year followed by two digits for the month and then two digits for the  
1467 day (e.g., 20180405 means the year 2018, month April, day 5<sup>th</sup>)
- 1468 - Any other character can be used in the mask, meaning simply that it appears in the same position; for  
1469 example, YYYY-MM-DD means that the values of year, month and day are separated by a line (e.g., 2018-  
1470 04-05 means the year 2018, month April, day 5<sup>th</sup>) and \PMM denotes the letter "P" followed by two  
1471 characters for the month.
- 1472 - The special characters and the special words, if prefixed by the reverse slash (\) in the mask, appear in the  
1473 same position in the time format; for example \PMMM means the letter "P" followed by two characters for  
1474 the month and then the letter "M"; for example, P03M means a period of three months (this is an ISO 8601  
1475 standard representation for a period of MM months). The reverse slash can appear in the format if needed  
1476 by prefixing it with another reverse slash; for example YYYY\\MM means for digits for the year, a backslash  
1477 and two digits for the month.

1478 -

1479 The **special characters** and the corresponding time units are the following:

1480	C	century
1481	Y	year
1482	S	semester
1483	Q	quarter
1484	M	month
1485	W	week
1486	D	day
1487	h	hour digit (by default on 24 hours)
1488	m	minute
1489	s	second
1490	d	decimal of second
1491	P	period indicator (see the "duration" codes below)
1492	p	number of periods

1493

1494 The **special words** for textual representations are the following:

1495 AM/PM indicator of AM / PM (e.g. am/pm for “am” or “pm”)  
 1496 MONTH textual representation of the month (e.g., JANUARY for January)  
 1497 DAY textual representation of the day (e.g., MONDAY for Monday)

1498

1499 **Examples of formatting masks for the *time* scalar type:**

1500 A Scalar Value of type *time* denotes time intervals of any duration and expressed with any precision, which are  
 1501 the intervening time between two time points.

1502 These examples are about three possible ISO 8601 formats for expressing time intervals:

- 1503 • Start and end time points, such as "2015-03-03T09:30:45Z/2018-04-05T12:30:15Z"
- 1504 VTL Mask: YYYY-MM-DDThh:mm:ssZ/YYYY-MM-DDThh:mm:ssZ
- 1505 • Start and duration, such as "2015-03-03T09:30:45-01/P1Y2M10DT2H30M"
- 1506 VTL Mask: YYYY-MM-DDThh:mm:ss-01/PY\YM\MDD\DT{h}h\HmM\M
- 1507 • Duration and end, such as "P1Y2M10DT2H30M/2018-04-05T12:30:00+02"
- 1508 VTL Mask: PY\YM\MDD\DT{h}h\HmM\M/YYYY-MM-DDThh:mm:ssZ

1509 Example of other possible ISO formats having accuracy reduced to the day

- 1510 • Start and end, such as "20150303/20180405"
- 1511 VTL Mask: YYYY-MM-DD/YYYY-MM-DD
- 1512 • Start and duration, such as "2015-03-03/P1Y2M10D"
- 1513 VTL Mask: YYYY-MM-DD/PY\YM\MDD\D
- 1514 • Duration and end, such as "P1Y2M10D/2018-04-05"
- 1515 VTL Mask: PY\YM\MDD\DT/YYYY-MM-DD

1516

1517 **Examples of formatting masks for the *date* scalar type:**

1518 A *date* scalar type is a point in time, equivalent to an interval of time having coincident start and end duration  
 1519 equal to zero.

1520 These examples about possible ISO 8601 formats for expressing dates:

- 1521 • Date and day time with separators: "2015-03-03T09:30:45Z"
- 1522 VTL Mask: YYYY-MM-DDThh:mm:ssZ
- 1523 • Date and day time without separators "20150303T093045-01 "
- 1524 VTL Mask: YYYYMMDDThhmmss-01

1525 Example of other possible ISO formats having accuracy reduced to the day

- 1526 • Date and day-time with separators "2015-03-03/2018-04-05"
- 1527 VTL Mask: YYYY-MM-DD/YYYY-MM-DD
- 1528 • Start and duration, such as "2015-03-03/P1Y2M10D"
- 1529 VTL Mask: YYYY-MM-DD/PY\YM\MDD\D

1530

1531 **Examples of formatting masks for the *time\_period* scalar type:**

1532 A *time\_period* denotes non-overlapping time intervals having a regular duration (for example the years, the  
 1533 quarters of years, the months, the weeks and so on). The *time\_period* values include the representation of the  
 1534 duration of the period.

1535 These examples are about possible formats for expressing time-periods:

- 1536 • Generic time period within the year such as: "2015Q4", "2015M12""2015D365"
- 1537 VTL Mask: YYYY{ppp} where P is the period indicator and ppp three digits for the number of
- 1538 periods, in the values, the period indicator may assume one of the values of the duration scalar type
- 1539 listed below.
- 1540 • Monthly period: "2015M03"
- 1541 VTL Mask: YYYY\MMM

1542

1543 **Examples of formatting masks for the *duration* scalar type:**

1544 A Scalar Value of type *duration* denotes the length of a time interval expressed with any precision and without

1545 connection to any particular time point (for example one year, half month, one hour and fifteen minutes).

1546 These examples are about possible formats for expressing durations (period / frequency)

1547 • Non ISO representation of the *duration* in one character, whose possible codes are:

1548	<i>Code</i>	<i>Duration</i>
1549	D	Day
1550	W	Week
1551	M	Month
1552	Q	Quarter
1553	S	Semester
1554	A	Year

1555 VTL Mask: P (period indicator)

1556 • ISO 8601 composite duration: "P10Y2M12DT02H30M15S" (P stands for "period")

1557 VTL Mask: \PYY\YM\MDD\DThh\Hmm\Mss\S

1558 • ISO 8601 duration in weeks: "P018W" (P stands for "period")

1559 VTL Mask: \PWWW\W

1560 • ISO 4 characters representation: P10M (ten months), P02Q (two quarters) ...

1561 VTL Mask: \PppP

1562

1563 Examples of fixed characters used in the ISO 8601 standard which can appear as fixed characters in the relevant

1564 masks:

1565	P	designator of duration
1566	T	designator of time
1567	Z	designator of UTC zone
1568	"+"	designator of offset from UTC zone
1569	"-"	designator of offset form UTC zone
1570	/	time interval separator

1571

## 1572 Attribute propagation

1573 The VTL has different default behaviours for Attributes and for Measures, to comply as much as possible with the

1574 relevant manipulation needs. At the Data Set level, the VTL Operators manipulate by default only the Measures

1575 and not the Attributes. At the Component level, instead, Attributes are calculated like Measures, therefore the

1576 algorithms for calculating Attributes, if any, can be specified explicitly in the invocation of the Operators. This is

1577 the behaviour of clauses like **calc**, **keep**, **drop**, **rename** and so on, either inside or outside the join (see the

1578 detailed description of these operators in the Reference Manual).

1579 The users which want to automatize the propagation of the Attributes' Values can optionally enforce a

1580 mechanism, called Attribute Propagation rule, whose behaviour is explained in the User Manual (see the section

1581 "Behaviour for Attribute Components"). The adoption of this mechanism is optional, users are free to allow the

1582 attribute propagation rule or not. The users that do not want to allow Attribute propagation rules simply will not

1583 implement what follows.

1584 In short, the automatic propagation of an Attribute depends on a Boolean characteristic, called "virality", which

1585 can be assigned to any Attribute of a Data Set (a viral Attribute has virality = TRUE, a non-viral Attribute has

1586 virality=FALSE, if the virality is not defined, the Attribute is considered as non-viral).

1587 By default, an Attribute propagates from the operand Data Sets (DS\_i) to the result Data Set (DS\_r) if it is "viral"

1588 at least in one of the operand Data Sets. By default, an Attribute which is viral in one of the operands DS\_i is

1589 considered as viral also in the result DS\_r.

1590 The Attribute propagation rule does not apply for the time series operators.  
1591 The Attribute propagation rule does not apply if the operations on the Attributes to be propagated are explicitly  
1592 specified in the expression (for example through the **keep** and **calc** operators). This way it is possible to keep in  
1593 the result also Attribute which are non-viral in all the operands, to drop viral Attributes, to override the  
1594 (possible) default calculation algorithm of the Attribute, to change the virality of the resulting Attributes.  
1595  
1596  
1597

1599 **Parentheses :**        **( )**

1600

1601 *Syntax*1602        **( op )**

1603

1604 *Input parameters*

1605 **op**        the operand to be evaluated before performing other operations written outside the parentheses.  
 1606            According to the general VTL rule, operators can be nested, therefore any Data Set, Component or scalar  
 1607            **op** can be obtained through an expression as complex as needed (for example **op** can be written as the  
 1608            expression **2 + 3** ).

1609

1610 *Examples of valid syntaxes*

1611 **( DS\_1 + DS\_2 )**  
 1612 **( CMP\_1 - CMP\_2 )**  
 1613 **( 2 + DS\_1 )**  
 1614 **( DS\_2 - 3 \* DS\_3 )**

1615

1616 *Semantic for scalar operations*

1617 Parentheses override the default evaluation order of the operators that are described in the section “VTL-ML –  
 1618 Evaluation order of the Operators”. The operations enclosed in the parentheses are evaluated first. For example  
 1619 **(2+3)\*4** returns 20, instead **2+3\*4** returns 14 because the multiplication has higher precedence than the  
 1620 addition.

1621

1622 *Input parameters type*

1623 **op ::**            dataset  
 1624                    | component  
 1625                    | scalar

1626

1627 *Result type*

1628 **result ::**        dataset  
 1629                    | component  
 1630                    | scalar

1631

1632 *Additional constraints*

1633 None.

1634

1635 *Behaviour*

1636 As mentioned, the **op** of the parentheses can be obtained through an expression as complex as needed (for  
 1637 example **op** can be written as **DS\_1 - DS\_2**. The part of the expression inside the parentheses is evaluated  
 1638 before the part outside of the parentheses. If more parentheses are nested, the inner parentheses are evaluated  
 1639 first, for example **( 20 – 10 / ( 2 + 3 ) ) \* 3** would give 54.

1640

1641 *Examples*

1642 **(DS\_1 + DS\_2) \* DS\_3**  
 1643 **(CMP\_1 – CMP\_2 / (CMP\_3 + CMP\_4) ) \* CMP\_5**

1644 **Persistent assignment :**        **<-**

1645

1646 *Syntax*1647        **re <- op**

1648

1649 *Input Parameters*  
1650 re the result  
1651 op the operand. According to the general VTL rule allowing the indentation of the operators, op can be  
1652 obtained through an expression as complex as needed (for example op can be the expression DS\_1 -  
1653 DS\_2).

1654 *Examples of valid syntaxes*  
1655 DS\_r <- DS\_1  
1656 DS\_r <- DS\_1 - DS\_2

1657 *Semantics for scalar operations*  
1658 empty

1659 *Input parameters type*  
1660 re :: name  
1661 op :: dataset

1662 *Result type*  
1663 empty

1664 *Additional constraints*  
1665 The assignment cannot be used at Component level because the result of a Transformation cannot be a Data Set  
1666 Component. When operations at Component level are invoked, the result is the Data Set which the output  
1667 Components belongs to.

1668 *Behaviour*  
1669 The input operand op is assigned to the **persistent** result re, which assumes the same value as op. As mentioned,  
1670 the operand op can be obtained through an expression as complex as needed (for example op can be the  
1671 expression DS\_1 - DS\_2).  
1672 The result re is a persistent Data Set that has the same data structure as the Operand. For example in DS\_r <-  
1673 DS\_1 the data structure of DS\_r is the same as the one of DS\_1.  
1674 If the Operand op is a scalar value, the result Data Set has no Components and contains only such a scalar value.  
1675 For example, income <- 3 assigns the value 3 to the persistent Data Set named income.

1676 *Examples*

1677 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

1687 *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

1690 Non-persistent assignment : :=

1691 *Syntax*

1692 re := op

1693 *Input parameters*

1695 re the result

1696 op the operand (according to the general VTL rule allowing the indentation of the operators, op can be  
1697 obtained through an expression as complex as needed (for example op can be the expression DS\_1 -  
1698 DS\_2).

1700 *Examples of valid syntaxes*

1701 DS\_r := DS\_1

1702 DS\_r := 3

1703 DS\_r := DS\_1 - DS\_2

1704 DS\_r := 3 + 2

1706 *Semantic for scalar operations*

1707 empty

1709 *Input parameters type*

1710 re :: name

1711 op :: dataset | scalar

1713 *Result type*

1714 empty

1716 *Additional constraints*

1717 The assignment cannot be used at Component level because the result of a Transformation cannot be a Data Set  
1718 Component. When operations at Component level are invoked, the result is the Data Set which the output  
1719 Components belongs to.

1720 The same symbol denoting the non-persistent assignment Operator (:=) is also used inside other operations at  
1721 Component level (for example in **calc** and **aggr**) in order to assign the result of the operation to the output  
1722 Component: please note that in these cases the symbol := does not denote the non-persistent assignment (i.e.,  
1723 this Operator), which cannot operate at Component level, but a special keyword of the syntax of the other  
1724 Operator in which it is used.

1726 *Behaviour*

1727 The value of the operand op is assigned to the result re, which is non-persistent and therefore is not stored. As  
1728 mentioned, the operand op can be obtained through an expression as complex as needed (for example op can be  
1729 the expression DS\_1 - DS\_2).

1730 The result re is a non-persistent Data Set that has the same data structure as the Operand. For example in DS\_r  
1731 := DS\_1 the data structure of DS\_r is the same as the one of DS\_1.

1732 If the Operand op is a scalar value, the result Data Set has no Components and contains only such a scalar value.  
1733 For example, income := 3 assigns the value 3 to the non-persistent Data Set named income.

1735 *Examples*

1736 Given the operand Data Sets DS\_1:

1738

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

1739

1740 Example 1: DS\_r := DS\_1 results in:  
1741

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

1742

1743 Membership : #

1744

1745 Syntax

1746 ds#comp

1747

1748 Input Parameters

1749 ds the Data Set

1750 comp the Data Set Component

1751

1752 Examples of valid syntaxes

1753 DS\_1#COMP\_3

1754

1755 Semantic for scalar operations

1756 This operator cannot be applied to scalar values.

1757

1758 Input parameters type

1759 ds :: dataset

1760 comp :: name < component >

1761

1762 Result type

1763 result :: dataset

1764

1765 Additional constraints

1766 comp must be a Data Set Component of the Data Set ds

1767

1768 Behaviour

1769 The membership operator returns a Data Set having the same Identifier Components of ds and a single Measure.

1770 If comp is a Measure in ds, then comp is maintained in the result while all other Measures are dropped.

1771 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).

1776 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.

1777 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.

1782

1783

1784 Examples

1785 Given the operand Data Set DS\_1:

1786



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	

1787  
 1788 *Example 1:*      DS\_r := DS\_1#Me\_1                      results in:  
 1789  
 1790 (assuming that At\_1 is not viral in DS\_1)  
 1791

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

1792  
 1793 (assuming that At\_1 is viral in DS\_1)  
 1794

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

1795  
 1796 *Example 2:*      DS\_r := DS\_1#Id\_1      assuming that At\_1 is viral in DS\_1 results in:  
 1797

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

1798  
 1799 *Example 3:*      DS\_r := DS\_1#At\_1      assuming that At\_1 is viral in DS\_1 results in:  
 1800

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

1801

## 1802 User-defined operator call

1803  
 1804 *Syntax*  
 1805      operatorName ( { argument { , argument }\* } )  
 1806

1807 *Input parameters*  
 1808 **operatorName** the name of an existing user-defined operator  
 1809 **argument** argument passed to the operator  
 1810  
 1811 *Examples of valid syntaxes*  
 1812 **max1 ( 2, 3 )**  
 1813  
 1814 *Semantic for scalar operations*  
 1815 It depends on the specific user-defined operator that is invoked.  
 1816  
 1817 *Input parameters type*  
 1818 **operatorName ::** name  
 1819 **argument ::** A data type compatible with the type of the parameter of the user-defined operator that  
 1820 is invoked (see also the “Type syntax” section).  
 1821  
 1822  
 1823 *Result type*  
 1824 **result ::** The data type of the result of the user-defined operator that is invoked (see also the  
 1825 “Type syntax” section).  
 1826  
 1827 *Additional constraints*  
 1828 • **operatorName** must refer to an operator created with the **define operator** statement.  
 1829 • The type of each argument value must be compliant with the type of the corresponding parameter of the  
 1830 user defined operator (the correspondence is in the positional order).  
 1831  
 1832 *Behaviour*  
 1833 The invoked user-defined operator is evaluated. The arguments passed to the operator in the invocation are  
 1834 associated to the corresponding parameters in positional order, the first argument as the value of the first  
 1835 parameter, the second argument as the value of the second parameter, and so on. An underscore (“\_”) can be  
 1836 used to denote that the value for an optional operand is omitted. One or more optional operands in the last  
 1837 positions can be simply omitted.  
 1838  
 1839 *Examples*  
 1840 *Example 1:*  
 1841  
 1842 Definition of the **max1** operator (see also “define operator” in the VTL-DL):  
 1843  
 1844       **define operator max1 (x integer, y integer)**  
 1845       **returns boolean**  
 1846       **is if x > y then x else y**  
 1847       **end define operator**  
 1848  
 1849 User-defined operator call of the **max1** operator:  
 1850  
 1851       **max1 ( 2, 3 )**  
 1852

## 1853 Evaluation of an external routine : **eval**

### 1854 *Syntax*

1856 **eval ( externalRoutineName ( { argument } { , argument }\* ) language languageName returns outputType )**  
 1857

### 1858 *Input parameters*

1859 **externalRoutineName** the name of an external routine  
 1860 **argument** the arguments passed to the external routine  
 1861 **language** the implementation language of the routine  
 1862 **outputType** the data type of the object returned by **eval** (see **outputParameterType** in Data  
 1863 type syntax)

1864  
1865  
1866  
1867  
1868  
1869  
1870  
1871  
1872  
1873  
1874  
1875  
1876  
1877  
1878  
1879  
1880  
1881  
1882  
1883  
1884  
1885  
1886  
1887  
1888  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907  
1908  
1909  
1910  
1911  
1912

### *Examples of valid syntaxes*

`eval ( routine1 ( "eabcdefgh" ) language "PL/SQL" returns string )`

### *Semantics for scalar operations:*

This is not a scalar operation.

### *Input parameters type*

<code>externalRoutineName ::</code>	name
<code>argument ::</code>	any data type
<code>language ::</code>	string
<code>outputType ::</code>	any data type restricting Data Set or scalar

### *Result Type*

<code>result ::</code>	dataset
------------------------	---------

### *Additional constraints*

- The **eval** is the only VTL Operator that does not allow nesting 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 ) language "PL/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 ) language "Java" returns integer ) ]
```

## 1913 Type conversion : **cast**

### *Syntax*

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

### *Input parameters*

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

1920

### 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 explicit casting:

- **“Explicit with mask”**: the explicit conversion uses a formatting mask that specifies how the actual casting is performed;
- **“Explicit w/o mask”**: the explicit conversion does not use 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 →	<i>integer</i>	<i>number</i>	<i>boolean</i>	<i>time</i>	<i>date</i>	<i>time_period</i>	<i>string</i>	<i>duration</i>
Provided								

<b>integer</b>	-	Implicit	Explicit w/o mask	Not feasible	Not feasible	Not feasible	Implicit	Not feasible
<b>number</b>	Explicit w/o mask	-	Explicit w/o mask	Not feasible	Not feasible	Not feasible	Implicit	Not feasible
<b>boolean</b>	Explicit w/o mask	Explicit w/o mask	-	Not feasible	Not feasible	Not feasible	Implicit	Not feasible
<b>time</b>	Not feasible	Not feasible	Not feasible	-	Not feasible	Not feasible	Explicit with mask	Not feasible
<b>date</b>	Not feasible	Not feasible	Not feasible	Implicit	-	Explicit w/o mask	Explicit with mask	Not feasible
<b>time_period</b>	Not feasible	Not feasible	Not feasible	Implicit	Explicit with mask	-	Explicit w/o mask	Not feasible
<b>string</b>	Explicit w/o mask	Explicit with mask	Not feasible	Explicit with mask	Explicit with mask	Explicit with mask	-	Explicit with mask
<b>duration</b>	Not feasible	Not feasible	Not feasible	Not feasible	Not feasible	Not feasible	Explicit with mask	-

1974

1975 The type of casting can be personalised in specific environments, provided that the personalisation is explicitly  
1976 documented with reference to the table above. For example, assuming that an explicit **cast** with **mask** is  
1977 required and that in a specific environment a definite **mask** is used for such a kind of conversions, the **cast** can  
1978 also become implicit provided that the **mask** that will be applied is specified.

1979 The **implicit casting** is performed when a value of a certain type is provided when another type is expected. Its  
1980 behaviour is described here:

- 1981 • From **integer** to **number**: an *integer* is provided when a *number* is expected (for example, an *integer* and a  
1982 *number* are passed as inputs of a n-ary numeric operator); it returns a *number* having the integer part equal  
1983 to the *integer* and the decimal part equal to zero;
- 1984 • From **integer** to **string**: an *integer* is provided when a *string* is expected (for example, an *integer* is passed  
1985 as an input of a *string* operator); it returns a *string* having the literal value of the *integer*;
- 1986 • From **number** to **string**: a *number* is provided when a *string* is expected; it returns the *string* having the  
1987 literal value of the *number*; the decimal separator is converted into the character "." (dot).
- 1988 • From **boolean** to **string**: a *boolean* is provided when a *string* is expected; the boolean value TRUE is  
1989 converted into the *string* "TRUE" and FALSE into the *string* "FALSE";
- 1990 • From **date** to **time**: a *date* (point in time) is provided when a *time* is expected (interval of time): the  
1991 conversion results in an interval having the same start and end, both equal to the original *date*;
- 1992 • From **time\_period** to **time**: a *time\_period* (a regular interval of *time*, like a month, a quarter, a year ...) is  
1993 provided when a *time* (any interval of time) is expected; it returns a *time* value having the same start and  
1994 end as the *time\_period* value.

1995 An implicit cast is also performed from a **value domain type** or a **set type** to a **basic scalar type**: when a *scalar*  
1996 value belonging to a Value Domains or a Set is involved in an operation (i.e., provided as input to an operator),  
1997 the value is implicitly cast into the basic scalar type which the Value Domain refers to (for this relationship, see  
1998 the description of Type System in the User Manual). For example, assuming that the Component *birth\_country* is  
1999 defined on the Value Domain *country*, which contains the ISO 3166-1 numeric codes and therefore refers to the  
2000 basic scalar type *integer*, the (possible) invocation *length(birth\_country)*, which calculates the length of the input  
2001 string, automatically casts the values of *birth\_country* into the corresponding string. If the basic scalar type of the  
2002 Value Domain is not compatible with the expression where it is used, an error is raised. This VTL feature is  
2003 particularly important as it provides a general behaviour for the Value Domains and relevant Sets, preventing  
2004 from the need of defining specific behaviours (or methods or operations) for each one of them. In other words,  
2005 all the Values inherit the operations that can be performed on them from the basic scalar types of the respective  
2006 Value Domains.

2007 The **cast** operator can be invoked explicitly even for the conversions which allow an implicit cast and in this case  
2008 the same behaviour as the implicit cast is applied.

2009 The behaviour of the **cast** operator for the conversions that require **explicit casting without mask** is the  
2010 following:

- 2011 • From **integer** to **boolean**: if the *integer* is different from 0, then TRUE is returned, FALSE otherwise.

- 2012 • From **number** to **integer**: converts a *number* with no decimal part into an *integer*; if the decimal part is  
2013 present, a runtime error is raised.
- 2014 • From **number** to **boolean**: if the *number* is different from 0.0, then TRUE is returned, FALSE otherwise.
- 2015 • From **boolean** to **integer**: TRUE is converted into 1; FALSE into 0.
- 2016 • From **boolean** to **number**: TRUE is converted into 1.0; FALSE into 0.0.
- 2017 • From **date** to **time\_period**: it converts a *date* into the corresponding daily value of *time\_period*.
- 2018 • From **string** to **integer**: the *integer* having the literal value of the *string* is returned; if the *string* contains a  
2019 literal that cannot be matched to an *integer*, a runtime error is raised.
- 2020 • From **string** to **time\_period**: it converts a *string* value to a *time\_period* value.

2021 When an **explicit casting with mask** is required, the conversion is made by applying the formatting mask which  
2022 specifies the meaning of the characters in the output *string*. The formatting Masks are described in the section  
2023 “VTL-ML – Typical Behaviour of the ML Operators”, sub-section “Type Conversion and Formatting Mask.

2024 The behaviour of the **cast** operator for such conversions is the following:

- 2025 • From **time** to **string**: it is applied the *time* formatting mask.
- 2026 • From **date** to **string**: it is applied the *time\_period* formatting mask.
- 2027 • From **time\_period** to **date**: it is applied a formatting mask which accepts two possible values (“START”,  
2028 “END”). If “START” is specified, then the *date* is set to the beginning of the *time\_period*; if “END” is specified,  
2029 then the *date* is set to the end of the *time\_period*.
- 2030 • From **time\_period** to **string**: it is applied the *time\_period* formatting mask.
- 2031 • From **duration** to **string**: a *duration* (an absolute time interval) is provided when a *string* is expected; it  
2032 returns the *string* having the default *string* representation for the *duration*.
- 2033 • From **string** to **number**: the *number* having the literal value of the *string* is returned; if the *string* contains a  
2034 literal that cannot be matched to a *number*, a runtime error is raised. The *number* is generated by using a  
2035 *number* formatting mask.
- 2036 • From **string** to **time**: the *time* having the literal value of the *string* is returned; if the *string* contains a literal  
2037 that cannot be matched to a *date*, a runtime error is raised. The *time* value is generated by using a *time*  
2038 formatting mask.
- 2039 • From **string** to **duration**: the *duration* having the literal value of the *string* is returned; if the *string* contains  
2040 a literal that cannot be matched to a *duration*, a runtime error is raised. The *duration* value is generated by  
2041 using a *time* formatting mask.

## 2042 Conversions between basic scalar types and Value Domains or Set types

2043 A value of a basic *scalar* type can be converted into a value belonging to a Value Domain which refers to such a  
2044 *scalar* type. The resulting *scalar* value must be one of the allowed values of the Value Domain or Set; otherwise, a  
2045 runtime error is raised. This specific use of **cast** operators does not really correspond to a type conversion; in  
2046 more formal terms, we would say that it acts as a constructor, i.e., it builds an instance of the output type. Yet,  
2047 towards a homogeneous and possibly simple definition of VTL syntax, we blur the distinction between  
2048 constructors and type conversions and opt for a unique formalism. An example is given below.

## 2049 Conversions between different Value Domain types

2050 As a result of the above definitions, conversions between values of different Value Domains are also possible.  
2051 Since an element of a Value Domain is implicitly cast into its corresponding basic scalar type, we can build on it  
2052 to turn the so obtained scalar type into another Value Domain type. Of course, this latter Value Domain type must  
2053 use as a base type this scalar type.

2054

## 2055 Examples

2056

2057 Example 1: from *string* to *number*

2058 `ds2 := ds1[calc m2 := cast(m1, number, “DD.DDD”) + 2 ]`

2059 In this case we use explicit cast from *string* to *numbers*. The mask is used to specify how the *string* must be  
2060 interpreted in the conversion.

2061

2062 Example 2: from *string* to *date*

2063 `ds2 := ds1[calc m2 := cast(m1, date, “YYYY-MM-DD”) ]`

2064 In this case we use explicit cast from *string* to *date*. The mask is used to specify how the *string* must be  
2065 interpreted in the conversion.  
2066

2067 Example 3: from *number* to *integer*  
2068 `ds2 := ds1[calc m2 := cast(m1, integer) + 3 ]`  
2069 In this case we cast a *number* into an *integer*, no mask is required.  
2070

2071 Example 4: from *number* to *string*  
2072 `ds2 := ds1[calc m2 := length(cast(m1, string)) ]`  
2073 In this case we cast a *number* into a *string*, no mask is required.  
2074

2075 Example 5: from *date* to *string*  
2076 `ds2 := ds1[calc m2 := cast(m1, string, "YY-MON-DAY hh:mm:ss") ]`  
2077 In this example a *date* instant is turned into a *string*. The mask is used to specify the *string* layout.  
2078

2079 Example 6: from *string* to *GEO\_AREA*  
2080 `ds2 := ds1[calc m2 := cast(GEO_STRING, GEO_AREA)]`  
2081 In this example we suppose we have elements of Value Domain Subset for *GEO\_AREA*. Let *GEO\_STRING* be a  
2082 string Component of Data Set *ds1* with string values compatible with the *GEO\_AREA* Value Domain Subset.  
2083 Thus, the following expression moves *ds1* data into *ds2*, explicitly casting strings to geographical areas.  
2084

2085 Example 7: from *GEO\_AREA* to *string*  
2086 `ds2 := ds1[calc m2 := length(GEO_AREA)]`  
2087 In this example we use a Component *GEO\_AREA* in a *string* expression, which calculates the length of the  
2088 corresponding *string*; this triggers the automatic cast.  
2089

2090 Example 8: from *GEO\_AREA2* to *GEO\_AREA1*  
2091 `ds2 := ds1 [ calc m2 := cast (GEO, GEO_AREA1) ]`  
2092 In this example we suppose we have to compare elements two Value Domain Subsets, They are both defined on  
2093 top of Strings. The following cast expressions performs the conversion.  
2094 Now, Component *GEO* is of type *GEO\_AREA2*, then we specify it has to be cast into *GEO\_AREA1*. As both  
2095 work on *strings* (and the values are compatible), the conversion is feasible. In other words, the cast of an  
2096 operand into *GEO\_AREA1* would expect a *string*. Then, as *GEO* is of type *GEO\_AREA2*, defined on top of  
2097 *strings*, it is implicitly cast to the respective *string*; this is compatible with what cast expects and it is then able to  
2098 build a value of type *GEO\_AREA1*.  
2099

2100 Example 9: from *string* to *time\_period*  
2101 In the following examples we convert from *strings* to *time\_periods*, by using appropriate masks.  
2102 The first quarter of year 2000 can be expressed as follows (other examples are possible):  
2103 `cast ( "2000Q1", time_period, "YYYY\QQ" )`  
2104 `cast ( "2000-Q1", time_period, "YYYY-\QQ" )`  
2105 `cast ( "2000-1", time_period, "YYYY-Q" )`  
2106 `cast ( "Q1-2000", time_period, "\QQ-YYYY" )`  
2107 `cast ( "2000Q01", time_period, "YYYY\QQQ" )`  
2108 Examples of daily data:  
2109 `cast ( "2000M01D01", time_period, "YYYY\MMM\DDD" )`  
2110 `cast ( "2000.01.01", time_period, "YYYY\MM\DD" )`  
2111



## 2112 VTL-ML - Join operators

2113 The Join operators are fundamental VTL operators. They are part of the core of the language and allow to obtain  
 2114 the behaviour of the majority of the other non-core operators, plus many additional behaviours that cannot be  
 2115 obtained through the other operators.  
 2116 The Join operators are four, namely the `inner_join`, the `left_join`, the `full_join` and the `cross_join`. Because their  
 2117 syntax is similar, they are described together.

2118 Join : `inner_join`, `left_join`, `full_join`, `cross_join`

### 2119 Syntax

```
2120 joinOperator ( ds { as alias } { , ds { as alias } }* { using usingComp { , usingComp }* }
2121           { filter filterCondition }
2122           { apply applyExpr
2123             | calc calcClause
2124             | aggr aggrClause { groupingClause } }
2125           { keep comp { , comp }* | drop comp { , comp }* }
2126           { rename compFrom to compTo { , compFrom to compTo }* }
2127           )
```

2128 joinOperator ::= { `inner_join` | `left_join` | `full_join` | `cross_join` }<sup>1</sup>

2129 calcClause ::= { calcRole } calcComp := calcExpr  
 2130 { , { calcRole } calcComp := calcExpr }\*

2131 calcRole ::= { `identifier` | `measure` | `attribute` | `viral attribute` }<sup>1</sup>

2132 aggrClause ::= { aggrRole } aggrComp := aggrExpr  
 2133 { , { aggrRole } aggrComp := aggrExpr }\*

2134 aggrRole ::= { `measure` | `attribute` | `viral attribute` }<sup>1</sup>

2135 groupingClause ::= { **group by** groupingId { , groupingId }\*  
 2136 | **group except** groupingId { , groupingId }\*  
 2137 | **group all** conversionExpr }<sup>1</sup>  
 2138 { **having** havingCondition }

### 2141 Input parameters

2142 <u>joinOperator</u>	the Join operator to be applied
2143 ds	the Data Set operands (at least one must be present)
2144 alias	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.
2146 usingComp	component of the input Data Sets whose values have to match in the join (the <b>using</b> clause is allowed for the <b>left_join</b> only under certain constraints described below and is not allowed at all for the <b>full_join</b> and <b>cross_join</b> )
2149 filterCondition	a condition ( <i>boolean</i> 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)
2152 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 <i>numeric</i> measures m1 and m2, the clause <code>apply ds1 + ds2</code> would result in calculating <code>m1 := ds1#m1 + ds2#m1</code> and <code>m2 := ds1#m2 + ds2#m2</code>
2157 <u>calcClause</u>	clause that specifies the Components to be calculated, their roles and their calculation algorithms, to be applied on the joined and filtered Data Points.
2159 <u>calcRole</u>	the role of the Component to be calculated
2160 calcComp	the name of the Component to be calculated



2161	calcExpr	expression at component level, having only Components of the input Data Sets as operands, used to calculate a Component
2162		
2163	<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
2164		
2165		
2166	<u>aggrRole</u>	the role of the aggregated Component to be calculated; if omitted, the Measure role is assumed
2167		
2168	aggrComp	the name of the aggregated Component to be calculated; this is a dependent Component of the result (Measure or Attribute, not Identifier)
2169		
2170	aggrExpr	expression at component level, having only Components of the input Data Sets as operands, which invokes an aggregate operator (e.g. <b>avg</b> , <b>count</b> , <b>max</b> ... , see also the corresponding sections) to perform the desired aggregation. Note that the <b>count</b> operator is used in an <b>aggrClause</b> without parameters, e.g.:
2171		
2172		
2173		
2174		DS_1 [ aggr Me_1 := count ( ) group by Id_1 ) ]
2175	<u>groupingClause</u>	the following alternative grouping options:
2176		<b>group by</b> the Data Points are grouped by the values of the specified Identifiers (groupingId). The Identifiers not specified are dropped in the result.
2177		
2178		<b>group except</b> the Data Points are grouped by the values of the Identifiers not specified as groupingId. The specified Identifiers are dropped in the result.
2179		
2180		
2181		<b>group all</b> converts the values of an Identifier Component using conversionExpr and keeps all the resulting Identifiers.
2182		
2183	groupingId	Identifier Component to be kept (in the <b>group by</b> clause) or dropped (in the <b>group except</b> clause).
2184		
2185	conversionExpr	specifies a conversion operator (e.g. <b>time_agg</b> ) to convert an Identifier from finer to coarser granularity. The conversion operator is applied on an Identifier of the operand Data Set.
2186		
2187		
2188	havingCondition	a condition ( <i>boolean</i> 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.:
2189		
2190		
2191		
2192		
2193		
2194		
2195		
2196		
2197		sum ( ds group by id1 having count ( ) >= 10 )
2198	comp	dependent Component (Measure or Attribute, not Identifier) to be kept (in the <b>keep</b> clause) or dropped (in the <b>drop</b> clause)
2199		
2200	compFrom	the original name of the Component to be renamed
2201	compTo	the new name of the Component after the renaming
2202		
2203	<i>Examples of valid syntaxes</i>	
2204	inner_join ( ds1 as d1, ds2 as d2 using Id1, Id2	
2205	filter d1#Me1 + d2#Me1 <10	
2206	apply d1 / d2	
2207	keep Me1, Me2, Me3	
2208	rename Id1 to Id10, id2 to id20	
2209	)	
2210		
2211	left_join ( ds1 as d1, ds2 as d2	
2212	filter d1#Me1 + d2#Me1 <10	
2213	calc Me1 := d1#Me1 + d2#Me3	
2214	keep Me1	
2215	rename Id1 to Ident1, Me1 to Meas1	
2216	)	
2217		
2218	full_join ( ds1 as d1, ds2 as d2	
2219	filter d1#Me1 + d2#Me1 <10	

```

2220         aggr Me1 := sum(Me1), attribute At20 := avg(Me2)
2221         group by Id1, Id2
2222         having sum(Me3) > 0
2223     )
2224

```

## 2225 *Semantics for scalar operations*

2226 The join operator does not perform scalar operations.

## 2227 *Input parameters type*

```

2228 ds::                dataset
2229 alias ::            name
2230 usingId ::          name < component >
2231 filterCondition ::  component<boolean>
2232 applyExpr ::        dataset
2233 calcComp ::         name < component >
2234 calcExpr ::         component<scalar>
2235 aggrComp ::         name < component >
2236 aggrExpr ::         component<scalar>
2237 groupingId ::       name < identifier >
2238 conversionExpr ::   component<scalar>
2239 havingCondition ::  component<boolean>
2240 comp ::             name < component >
2241 compFrom ::         component<scalar>
2242 compTo ::           component<scalar>
2243
2244

```

## 2245 *Result type*

```

2246 result ::          dataset
2247

```

## 2248 *Additional constraints*

2249 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.

2251 The using clause is not allowed for the **full\_join** and for the **cross\_join**, because otherwise a non-functional result could be obtained.

2253 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<sup>7</sup>:

- 2255 • 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.
- 2257 • 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.
- 2259 • For **cross\_join** (Cartesian product), no constraints are needed.

2260 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:

- 2263 • 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;
- 2265 • Sub-case B2:
  - 2266 ○ In case of **inner\_join**, one Data Set acts as the reference Data Set which the others are joined to;
  - 2267 ○ in case of **left\_join**, this is the left-most Data Set (i.e.,  $ds_1$ );
  - 2268 ○ All the input Data Sets, except the reference Data Set, have the same Identifiers  $[Id_1, \dots, Id_n]$ ;
  - 2269 ○ The using clause specifies all and only the common Identifiers of the non-reference Data Sets
  - 2270  $[Id_1, \dots, Id_n]$ .

2271 The join operators must fulfil also other constraints:

- 2272 • **apply**, **calc** and **aggr** clauses are mutually exclusive
- 2273 • **keep** and **drop** clauses are mutually exclusive
- 2274 • **comp** can be only dependent Components (Measures and Attributes, not Identifiers)
- 2275 • An Identifier not included in the **group by** clause (if any) cannot be included in the **rename** clause

---

<sup>7</sup> 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

2281

## 2282 *Behaviour*

2283 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<sub>1</sub>).
- The filterCondition, if present, is applied on VDS<sub>1</sub>, producing the Virtual Data Set VDS<sub>2</sub>.
- The specified calculation algorithms (**apply**, **calc** or **aggr**), if present, are applied on VDS<sub>2</sub>. 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<sub>3</sub>.
- The **keep** or **drop** clause, if present, is applied on VDS<sub>3</sub>, producing the Virtual Data Set VDS<sub>4</sub>.
- The **rename** clause, if present, is applied on VDS<sub>4</sub>, producing the Virtual Data Set VDS<sub>5</sub>.
- The final automatic alias removal is performed in order to obtain the output Data Set.

2295 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.

2299 The **structure of the virtual Data Set** VDS<sub>1</sub> which is the output of the relational join is the following.

2300 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#cmp<sub>i</sub>). 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.

2321 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.

2323 The **semantics of the relational join** is the following.

2324 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.

2326 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.

2328 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.

2330 The different kinds of relational joins behave as follows.

- **inner\_join**: the Data Points of ds<sub>1</sub>, ..., ds<sub>N</sub> 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<sub>1</sub> 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  $\langle ds_i, ds_{i+1} \rangle$ , 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  $\langle ds_i, ds_{i+1} \rangle$ , 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  $\langle ds_i, ds_{i+1} \rangle$ , 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  $\langle ds_i, ds_{i+1} \rangle$ , 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  $\langle ds_i, ds_{i+1} \rangle$ , 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  $\langle ds_i, ds_{i+1} \rangle$ , 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 output. 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.

<b>group by</b>	the Data Points are grouped by the values of the specified Identifier. The Identifiers not specified are dropped in the result.
<b>group except</b>	the Data Points are grouped by the values of the Identifiers not specified in the clause. The specified Identifiers are dropped in the result.
<b>group all</b>	converts an Identifier Component using <code>conversionExpr</code> 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

2454 input Data Set have homonym Measures and there is no apply clause which unifies them; these Measures can be  
2455 renamed or removed.

2456  
2457 *Examples*

2458  
2459 Given the operand Data Sets DS\_1 and DS\_2:  
2460

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

2461

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

2462

2463

2464 *Example 1:*

2465 DS\_r := inner\_join ( DS\_1 as d1, DS\_2 as d2  
2466 keep Me\_1, d2#Me\_2, Me\_1A) results in:

2467

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

2468

2469 *Example 2:*

2470 DS\_r := left\_join ( DS\_1 as d1, DS\_2 as d2  
2471 keep Me\_1, d2#Me\_2, Me\_1A ) results in:

2472

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

2473

2474 *Example 3:*

2475 DS\_r := full\_join ( DS\_1 as d1, DS\_2 as d2  
2476 keep Me\_1, d2#Me\_2, Me\_1A ) results in:

2477

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:

2511

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

2512  
2513  
2514  
2515  
2516

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

2517  
2518  
2519



2520 **VTL-ML - String operators**

2521 **String concatenation :**                    **||**

2522  
2523 *Syntax*  
2524        op1 || op2

2525  
2526 *Input Parameters*  
2527 op1, op2            the operands

2528  
2529 *Examples of valid syntaxes*  
2530 "Hello" || ", world!"  
2531 ds\_1 || ds\_2

2532  
2533 *Semantics for scalar operations*  
2534 Concatenates two strings. For example, "Hello" || ", world!" gives "Hello, world!"

2535  
2536 *Input parameters type*  
2537 op1, op2 ::        dataset { measure<string> \_+ }  
2538                    | component<string>  
2539                    | string

2540  
2541 *Result type*  
2542 result ::            dataset { measure<string> \_+ }  
2543                    | component<string>  
2544                    | string

2545  
2546 *Additional constraints*  
2547 None.

2548  
2549 *Behaviour*  
2550 The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set  
2551 Components” (see the section “Typical behaviours of the ML Operators”).

2552  
2553 *Examples*  
2554 Given the Data\_Sets DS\_1 and DS\_2:

2555

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

2556  
2557

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

2558  
2559 *Example 1:* DS\_r := DS\_1 || DS\_2        results in:  
2560

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}<sup>1</sup> ( 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 "

2600 Example 1: DS\_r := rtrim(DS\_1) results in:  
2601

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

2602  
2603 Example 2 (on component): DS\_r := DS\_1[ calc Me\_2:= rtrim(Me\_1)] results in:  
2604

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

2605 Character case conversion : upper/lower

2606 Syntax  
2607 {upper | lower}<sup>1</sup> ( op )

2608  
2609 Input Parameters  
2610 op the operand

2611  
2612 Examples of valid syntaxes  
2613 upper("Hello")  
2614 lower(ds\_1)

2615  
2616 Semantics for scalar operations  
2617 Converts the character case of a string in upper or lower case. For example, upper("Hello") gives "HELLO".

2618  
2619 Input Parameters type  
2620 op :: dataset { measure<string> \_+ }  
2621 | component<string>  
2622 | string

2623  
2624 Result type  
2625 result :: dataset { measure<string> \_+ }  
2626 | component<string>  
2627 | string

2628  
2629 Additional constraints  
2630 None.

2631  
2632 Behaviour  
2633 The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set  
2634 Component” (see the section “Typical behaviours of the ML Operators”).

2635  
2636 Examples  
2637 Given the Data Set DS\_1:  
2638

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

2639  
2640  
2641

Example 1: DS\_r := upper(DS\_1)      results in:

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

2642  
2643  
2644

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"

2645

## 2646      Sub-string extraction :                      substr

2647  
2648  
2649

### Syntax

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

2650

### 2651      Input parameters

2652      op      the operand

2653      start      the starting digit (first character) of the string to be extracted

2654      length      the length (number of characters) of the string to be extracted

2655

### 2656      Examples of valid syntaxes

2657      substr ( DS\_1, 2 , 3 )

2658      substr ( DS\_1, 2 )

2659      substr ( DS\_1, \_ , 3 )

2660      substr ( DS\_1 )

2661

### 2662      Semantics for scalar operations

2663      The operator extracts a substring from op, which must be *string* type. The substring starts from the start<sup>th</sup>  
2664      character of the input string and has a number of characters equal to the length parameter.

- 2665      • If start is omitted, the substring starts from the 1<sup>st</sup> position.
- 2666      • If length is omitted or overcomes the length of the input string, the substring ends at the end of the input  
2667      string.
- 2668      • If start is greater than the length of the input string, an empty string is extracted.

2669

2670      For example:

2671      substr ( "abcdefghijklmnopqrstuvwxy", 5 , 10 )                      gives: "efghijklmn".

2672      substr ( "abcdefghijklmnopqrstuvwxy", 25 , 10 )                      gives: "yz".

2673      substr ( "abcdefghijklmnopqrstuvwxy", 30 , 10 )                      gives: "".

2674

### 2675      Input parameters type

2676      op ::                      dataset { measure <string> \_+ }

2677                                  | component <string>

2678                                  | string

2679

2680      start ::                      component < integer [ value >= 1 ] >

2681                                  | integer [ value >= 1 ]

2682

2683

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	"abcdefghijklmno"	"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	"abcdefghijklmno"	"short"

2	A	"pqrstuvwxyz"	"this "
---	---	---------------	---------

2716

2717 String pattern replacement: **replace**

2718 *Syntax*

2719 **replace** (op , pattern1, pattern2 )

2720

2721 *Input parameters*

2722 op            the operand

2723 pattern1    the pattern to be replaced

2724 pattern2    the replacing pattern

2725

2726 *Examples of valid syntaxes*

2727 replace(DS\_1, "Hello", "Hi")

2728 replace(DS\_1, "Hello")

2729

2730 *Semantics for scalar operations*

2731 Replaces all the occurrences of a specified string-pattern (pattern1) with another one (pattern2). If pattern2 is  
2732 omitted then all occurrences of pattern1 are removed. For example:

2733

2734 replace("Hello world", "Hello", "Hi")       gives "Hi world"

2735 replace("Hello world", "Hello")            gives " world"

2736 replace ("Hello", "ello", "i")            gives "Hi"

2737

2738 *Input parameters type*

2739 op ::                                dataset { measure<string> \_+ }

2740                                        | component<string>

2741                                        | string

2742 pattern1, pattern2 ::        component<string>

2743                                        | string

2744

2745 *Result type*

2746 result ::                        dataset { measure<string> \_+ }

2747                                        | component<string>

2748                                        | string

2749

2750 *Additional constraints*

2751 None.

2752

2753 *Behaviour*

2754 As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar  
2755 Value or Data Set or Data Set Component”, as for the invocations at Component or Scalar level, the operator has  
2756 the behaviour of the “Operators applicable on more than two Scalar Values or Data Set Components”, (see the  
2757 section “Typical behaviours of the ML Operators”).

2758

2759 *Examples*

2760 Given the Data\_Set DS\_1:

2761

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

2762

2763 Example 1: DS\_r := replace (ds\_1,"ello","i") results in:  
2764

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

2765  
2766 Example 2 (on component): DS\_r := DS\_1[ calc Me\_2:= replace (Me\_1,"ello","i")] results in:  
2767

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! "

2768

2769 String pattern location : instr

2770

2771 Syntax  
2772 instr ( op, pattern, start, occurrence )

2773

2774

2775 Input parameters

2776 op the operand  
2777 pattern the string-pattern to be searched  
2778 start the position in the input string of the character from which the search starts  
2779 occurrence the occurrence of the pattern to search

2780

2781 Examples of valid syntaxes

2782 instr ( DS\_1, "ab", 2 , 3 )  
2783 instr ( DS\_1, "ab", 2 )  
2784 instr ( DS\_1, "ab", \_ , 2 )  
2785 instr ( DS\_1, "ab" )

2786

2787 Semantics for scalar operations

2788 The operator returns the position in the input string of a specified string (pattern). The search starts from the  
2789 start<sup>th</sup> character of the input string and finds the n<sup>th</sup>occurrence of the pattern, returning the position of its first  
2790 character.

- 2791 • If start is omitted, the search starts from the 1<sup>st</sup> position.
- 2792 • If n<sup>th</sup>occurrence is omitted, the value is 1.

2793 If the n<sup>th</sup>occurrence of the string-pattern after the start<sup>th</sup> character is not found in the input string, the returned  
2794 value is 0.

2795

2796 For example:

2797 instr ("abcde", "c" ) gives 3  
2798 instr ("abcdecfrxcwsd", "c", \_ , 3 ) gives 10  
2799 instr ("abcdecfrxcwsd", "c", 5 , 3 ) gives 0

2800

2801 Input parameters type

```

2802 op ::      dataset { measure<string> _ }
2803           | component<string>
2804           | string
2805 pattern ::  component<string>
2806           | string
2807 start ::    component < integer [ value >= 1 ] >
2808           | integer [ value >= 1 ]
2809 occurrence :: component < integer [ value >= 1 ] >
2810           | integer [ value >= 1 ]
2811
2812 Result type
2813 result ::   dataset { measure<integer[value >= 0]> int_var }
2814           | component<integer[value >= 0]>
2815           | integer[value >= 0]
2816

```

### 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



2836  
2837  
2838  
2839

Given the Data Set DS\_2:

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

2840  
2841  
2842  
2843  
2844

*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

2845  
2846  
2847  
2848  
2849  
2850

*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.

2851

## String length :      length

2852  
2853  
2854  
2855  
2856  
2857  
2858  
2859  
2860  
2861

### *Syntax*

**length ( op )**

### *Input Parameters*

op      the operand

### *Examples of valid syntaxes*

length("Hello, World!")  
length(DS\_1)

2862  
2863  
2864  
2865

### *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

2866

### *Input Parameters type*

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

2870

### *Result type*

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

2875

2876

### *Additional constraints*

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

2878

2879

### *Behaviour*

2880 The operator has the behaviour of the “Operators changing the data type” (see the section “Typical behaviours of  
 2881 the ML Operators”).  
 2882 If op is a Data Set then **length** returns a dataset with a single measure int\_var of type *integer*.

2883  
 2884 *Examples*

2885  
 2886 Given the Data Set DS\_1  
 2887

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

2888  
 2889  
 2890 *Example 1:* DS\_r := length(DS\_1) results in:  
 2891

DS_r		
Id_1	Id_2	int_var
1	A	5
2	B	null

2892  
 2893 *Example 2 (on component):* DS\_r:= DS\_1[calc Me\_2:=length(Me\_1)] results in  
 2894

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

2895  
 2896 Given the Data Set DS\_2:  
 2897

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

2898  
 2899 *Example 3 (applying the **length** operator at component level to a multi Measure Data Set):*

2900  
 2901 DS\_r := DS\_2 [calc Me\_10:= length(Me\_1), Me\_20:=length(Me\_2)] results in:  
 2902

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

2903  
 2904 *Example 4 (**length** operator applied at Data Set level to a multi Measure Data Set):*

2905  
 2906 DS\_r := length(DS\_2) would give error because DS\_2 has more Measures.

2908 Unary plus :                    +

2909 *Syntax*

2910                    + op

2912 *Input parameters*

2913 op            the operand

2915 *Examples of valid syntaxes*

2916 + DS\_1

2917 + 3

2919 *Semantics for scalar operations*

2920 The operator + returns the operand unchanged. For example:

2921            + 3                    gives    3

2922            + ( - 5 )            gives   - 5

2924 *Input Parameters type*

2925 op ::            dataset { measure<number> \_+ }  
2926                    | component<number>  
2927                    | number

2929 *Result type*

2930 result ::            dataset { measure<number> \_+ }  
2931                    | component<number>  
2932                    | number

2934 *Additional constraints*

2935 None.

2937 *Behaviour*

2938 The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set  
2939 Component” (see the section “Typical behaviours of the ML Operators”).

2940 According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is  
2941 the type *integer*. If the type of the operand is *integer* then the result has type *integer*. If the type of the operand is  
2942 *number* then the result has type *number*.

2944 *Examples*

2945 Given the operand Data Set DS\_1:

2946

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

2947

2948 *Example 1:*                    DS\_r := + DS\_1                    results in:

2949

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

2950  
 2951 *Example 2 (on components):* DS\_r := DS\_1 [calc Me\_3 := + Me\_1 ]  
 2952

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

2953 **Unary minus:** -

2954 *Syntax*

2955 - op

2956 *Input parameters*

2957 op the operand

2958 *Examples of valid syntaxes*

2959 - DS\_1

2960 - 3

2961 *Semantics for scalar operations*

2962 The operator - inverts the sign of op. For example:

2963 - 3 gives - 3

2964 - ( - 5 ) gives 5

2965 *Input Parameters type*

2966 op :: dataset { measure<number> \_+ }

2967 | component<number>

2968 | number

2969 *Result type*

2970 result :: dataset { measure<number> \_+ }

2971 | component<number>

2972 | number

2973 *Additional constraints*

2974 None.

2975 *Behaviour*

2976 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”).

2977 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*.

2978 *Examples*

2979 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

2992  
2993  
2994

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

2995  
2996  
2997

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

2998  
2999

3000    Addition :                      +

3001    *Syntax*  
3002                      op1 + op2

3003  
3004    *Input parameters*  
3005    op1        the first addendum  
3006    op2        the second addendum

3007  
3008    *Examples of valid syntaxes*  
3009    DS\_1 + DS\_2  
3010    3 + 5

3011    *Semantics for scalar operations*  
3012  
3013    The operator addition returns the sum of two numbers. For example:  
3014                      3 + 5    gives    8

3015  
3016    *Input parameters type*  
3017    op1, op2 ::        dataset { measure<number> \_+ }  
3018                      | component<number>  
3019                      | number

3020  
3021    *Result type*  
3022    result ::            dataset { measure<number> \_+ }  
3023                      | component<number>  
3024                      | number

3025  
3026    *Additional constraints*  
3027    None.

3028

*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

3049 Subtraction : -

3050 *Syntax*  
3051 op1 - op2

3052  
3053 *Input Parameters*  
3054 op1 the minuend  
3055 op2 the subtrahend

3056  
3057 *Examples of valid syntaxes*  
3058 DS\_1 - DS\_2  
3059 3 - 5

3060  
3061 *Semantics for scalar operations*  
3062 The operator subtraction returns the difference of two numbers. For example:  
3063 3 - 5 gives - 2

3064  
3065 *Input Parameters type*  
3066 op1, op2:: dataset { measure<number> \_+ }  
3067 | component<number>  
3068 | number

3069  
3070 *Result type*  
3071 result :: dataset { measure<number> \_+ }  
3072 | component<number>  
3073 | number

3074  
3075 *Additional constraints*  
3076 None.

3077  
3078 *Behaviour*  
3079 The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set  
3080 Components” (see the section “Typical behaviours of the ML Operators”).  
3081 According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is  
3082 the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is  
3083 of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.

3084  
3085 *Examples*  
3086 Given the operand Data Sets DS\_1 and DS\_2:  
3087

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

3088

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

3089  
3090 *Example 1:* DS\_r := DS\_1 - DS\_2 results in:  
3091

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):*      $DS\_r := DS\_1 \ [ \text{calc } Me\_3 := Me\_1 * Me\_2 \ ]$      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

3149

3150

## Division : /

3151

### Syntax

3152

op1 / op2

3153

3154

### Input parameters

3155

op1 the dividend

3156

op2 the divisor

3157

3158

### Examples of valid syntaxes

3159

DS\_1 / DS\_2

3160

3 / 5

3161

3162

### Semantics for scalar operations

3163

The operator **division** divides two numbers. For example:

3164

3 / 5 gives 0.6

3165

3166

### Input parameters type

3167

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

3168

| component<number>

3169

| number

3170

3171

### Result type

3172

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

3173

| component<number>

3174

| number

3175

3176

### Additional constraints

3177

None.

3178

3179

### Behaviour

3180

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”).

3181

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*.

3182

If op2 is 0 then the operation generates a run-time error.

3183

3184

3185

3186

### Examples

3187

Given the operand Data Sets DS\_1 and DS\_2:

3188

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):       $DS_r := DS_1 \text{ [ calc Me}_3 := \text{Me}_2 / \text{Me}_1 \text{ ]}$       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*

3209 mod ( DS\_1, DS\_2 )  
3210 mod ( DS\_1, 5 )  
3211 mod ( 5, DS\_2 )  
3212 mod ( 5, 2 )  
3213

3214 *Semantics for scalar operations*

3215 The operator **mod** returns the remainder of op1 divided by op2. It returns op1 if divisor op2 is 0. For example:

3216 mod ( 5, 2 ) gives 1  
3217 mod ( 5, -2 ) gives -1  
3218 mod ( 8, 2 ) gives 0  
3219 mod ( 9, 0 ) gives 9  
3220

3221 *Input Parameters type*

3222 op1, op2 :: dataset { measure<number> \_+ }  
3223 | component<number>  
3224 | number  
3225 divisor :: number  
3226

3227 *Result type*

3228 result :: dataset { measure<number> \_+ }  
3229 | component<number>  
3230 | number  
3231

3232 *Additional constraints*

3233 None.  
3234

3235 *Behaviour*

3236 The operator has the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set  
3237 Components” (see the section “Typical behaviours of the ML Operators”).  
3238 According to the general rules about data types, the operator can be applied also on sub-types of *number*, that is  
3239 the type *integer*. If the type of both operands is *integer* then the result has type *integer*. If one of the operands is  
3240 of type *number*, then the other operand is implicitly cast to *number* and therefore the result has type *number*.  
3241

3242 *Examples*

3243 Given the operand Data Sets DS\_1 and DS\_2:  
3244

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

3246  
3247  
3248 *Example 1:* DS\_r := mod ( DS\_1, DS\_2 ) results in:  
3249

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

Example 2:  $DS\_r := \text{mod} ( DS\_1, 15 )$  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):  $DS\_r := DS\_1[ \text{calc } Me\_3 := \text{mod}( DS\_1\#Me\_1, 3.0 ) ]$  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 rounding 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

3282            round ( 12345.6 )            gives 12346  
 3283            round ( 12345.6, \_ )        gives 12346  
 3284            round ( 12345.6, -1 )       gives 12350.0

3285

#### 3286 *Input parameters type*

3287    op1 ::            dataset { measure<number> \_+ }  
 3288                      | component<number>  
 3289                      | number  
 3290    numDigit::        component < integer >  
 3291                      | integer

3292

#### 3293 *Result type*

3294    result ::        dataset { measure<number> \_+ }  
 3295                      | component<number>  
 3296                      | number

3297

#### 3298 *Additional constraints*

3299    None.

3300

#### 3301 *Behaviour*

3302    As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar  
 3303    Value or Data Set or Data Set Component”, as for the invocations at Component or Scalar level, the operator has  
 3304    the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components”, (see the  
 3305    section “Typical behaviours of the ML Operators”).

3306

#### 3307 *Examples*

3308    Given the operand Data Set DS\_1:

3309

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

3310

3311    *Example 1:*        DS\_r := round(DS\_1, 0)                      results in:

3312

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

3313

3314    *Example 2 (on components):*        DS\_r := DS\_1 [ calc Me\_10:= round( Me\_1 ) ]                      results in:

3315

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



3383     **Ceiling :            ceil**

3384     *Syntax*

3385             **ceil ( op )**

3386     *Input parameters*

3387     op        the operand

3389     *Examples of valid syntaxes*

3391     ceil ( DS\_1 )

3392     ceil ( 3.14159 )

3393     *Semantics for scalar operations*

3394     The operator **ceil** returns the smallest integer greater than or equal to op.

3395     For example:

3396             ceil( 3.14159 ) gives     4

3397             ceil( 15 )           gives    15

3398             ceil( -3.1415 ) gives    -3

3399             ceil( -0.1415 ) gives    0

3401     *Input parameters type*

3402     op ::                dataset { measure<number> \_+ }  
3403                            | component<number>  
3404                            | number

3406     *Result type*

3407     result ::            dataset { measure<integer> \_+ }  
3408                            | component< integer >  
3409                            | integer

3411     *Additional constraints*

3412     None.

3414     *Behaviour*

3415     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”).

3418     *Examples*

3419     Given the operand Data Set DS\_1:

3420     3421

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

3422     *Example 1:*        DS\_r := ceil (DS\_1)                    results in:

3423     3424

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
----	---	----	---

3425  
 3426 *Example 2 (on components):*      DS\_r := DS\_1 [ calc Me\_10 := ceil (Me\_1) ]  
 3427

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

3428

3429 **Floor:                      floor**

3430 *Syntax*  
 3431                      **floor ( op )**

3432  
 3433 *Input parameters*  
 3434 op            the operand

3435  
 3436 *Examples of valid syntaxes*  
 3437 floor ( DS\_1 )  
 3438 floor ( 3.14159 )

3439  
 3440 *Semantics for scalar operations*  
 3441 The operator **floor** returns the greatest integer which is smaller than or equal to op.  
 3442 For example:

3443            floor( 3.1415 ) gives    3  
 3444            floor( 15 )            gives    15  
 3445            floor( -3.1415 ) gives    -4  
 3446            floor( -0.1415 ) gives    -1

3447  
 3448 *Input parameters type*  
 3449 op ::                      dataset { measure<number> \_+ }  
 3450                              | component<number>  
 3451                              | number

3452  
 3453 *Result type*  
 3454 result ::                      dataset { measure<integer> \_+ }  
 3455                              | component< integer >  
 3456                              | integer

3457  
 3458 *Additional constraints*  
 3459 None.

3460  
 3461 *Behaviour*  
 3462 The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set  
 3463 Component” (see the section “Typical behaviours of the ML Operators”).

3464  
 3465 *Examples*  
 3466 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

3468  
 3469 *Example 1:* DS\_r := floor ( DS\_1 ) results in:  
 3470

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

3471  
 3472 *Example 2 (on components):* DS\_r := DS\_1 [ calc Me\_10 := floor (Me\_1) ] results in:  
 3473

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

3474 **Absolute value :**                    **abs**

3475 *Syntax*  
 3476                    **abs ( op )**

3477 *Input parameters*  
 3478 op            the operand

3480 *Examples of valid syntaxes*  
 3481 abs ( DS\_1 )  
 3482 abs ( -5 )

3484 *Semantics for scalar operations*  
 3485 The operator **abs** calculates the absolute value of a number.  
 3486 For example:

3488            abs ( -5.49 )    gives 5.49  
 3489            abs ( 5.49 )    gives 5.49

3490 *Input parameters type*

3491 op ::                    dataset { measure<number> \_+ }  
 3492                            | component<number>  
 3493                            | number

3494 *Result type*

3495 result ::                dataset { measure<number [ value >= 0 ]> \_+ }  
 3496                            | component<number [ value >= 0 ]>  
 3500

| 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 [ calc 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)

3536 exp ( 0 ) gives 1.0  
3537 exp ( -1 ) gives 0.36787... (the number 1/e)

3538  
3539 *Input parameters type*

3540 op:: dataset { measure<number> \_+ }  
3541 | component<number>  
3542 | number

3543  
3544 *Result type*

3545 result :: dataset { measure<number[value > 0]> \_+ }  
3546 | component<number [value > 0]>  
3547 | number[value > 0]

3548  
3549 *Additional constraints*

3550 None.

3551  
3552 *Behaviour*

3553 The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set  
3554 Component” (see the section “Typical behaviours of the ML Operators”).

3555  
3556 *Examples*

3557 Given the operand Data Set DS\_1:

3558

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

3559

3560

3561 *Example 1:* DS\_r := exp(DS\_1) results in:

3562

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

3563

3564 *Example 2 (on components):* DS\_r := DS\_1 [ calc Me\_1 := exp ( Me\_1 ) ]

3565 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

3566

3567 Natural logarithm : ln

3568 *Syntax*

3569 ln ( op )

3570  
3571  
3572  
3573  
3574  
3575  
3576  
3577  
3578  
3579  
3580  
3581  
3582  
3583  
3584  
3585  
3586  
3587  
3588  
3589  
3590  
3591  
3592  
3593  
3594  
3595  
3596  
3597  
3598  
3599  
3600  
3601  
3602  
3603  
3604  
3605  
  
3606  
3607  
3608  
3609  
  
3610  
3611  
3612

*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 [ calc 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

3613

3614 **Power :**            **power**

3615 *Syntax*

3616            **power ( base , exponent )**

3617

3618 *Input parameters*

3619 base            the operand

3620 exponent        the exponent of the power

3621

3622 *Examples of valid syntaxes*

3623 power ( DS\_1, 2 )

3624 power ( 5, 2 )

3625

3626 *Semantics for scalar operations*

3627 The operator **power** raises a number (the base) to another one (the exponent).

3628 For example:

3629            power ( 5, 2 ) gives 25

3630            power ( 5, 1 ) gives 5

3631            power ( 5, 0 ) gives 1

3632            power ( 5, -1 ) gives 0.2

3633            power ( -5, 3 ) gives -125

3634

3635 *Input parameters type*

3636 base ::            dataset { measure<number> \_+ }

3637                    | component<number>

3638                    | number

3639 exponent ::        component<number>

3640                    | number

3641

3642 *Result type*

3643 result ::            dataset { measure<number> \_+ }

3644                    | component<number>

3645                    | number

3646

3647 *Additional constraints*

3648 None.

3649

3650 *Behaviour*

3651 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”).

3655

3656 *Examples*

3657 Given the operand Data Set DS\_1:

3658

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



3696  
3697 *Additional constraints*  
3698 None.

3699 *Behaviour*  
3701 As for the invocations at Data Set level, the operator has the behaviour of the “Operators applicable on one Scalar  
3702 Value or Data Set or Data Set Component”, as for the invocations at Component or Scalar level, the operator has  
3703 the behaviour of the “Operators applicable on two Scalar Values or Data Sets or Data Set Components”, (see the  
3704 section “Typical behaviours of the ML Operators”).  
3705

3706 *Examples*  
3707 Given the operand Data Set DS\_1:  
3708

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

3709  
3710  
3711 *Example 1:* DS\_r := log ( DS\_1, 2 ) results in:  
3712

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

3713  
3714 *Example 2 (on components):* DS\_r := DS\_1 [ calc Me\_1 := log (Me\_1, 2) ] results in:  
3715

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

3716

3717 **Square root :** **sqrt**

3718 *Syntax*  
3719 **sqrt ( op )**

3720  
3721 *Input parameters*  
3722 op the operand

3723  
3724 *Examples of valid syntaxes*  
3725 sqrt ( DS\_1 )  
3726 sqrt ( 5 )

3727  
3728 *Semantics for scalar operations*  
3729 The operator **sqrt** calculates the square root of a number. For example:  
3730 sqrt ( 25 ) gives 5

3731  
3732 *Input parameters type*  
3733 op :: dataset { measure<number [value >= 0] > \_+ }  
3734 | component<number [value >= 0] >  
3735 | number [value >= 0]

3736  
3737 *Result type*  
3738 result :: dataset { measure<number[value >= 0] > \_+ }  
3739 | component<number[value >= 0] >  
3740 | number[value >= 0]

3741  
3742 *Additional constraints*  
3743 None.

3744  
3745 *Behaviour*  
3746 The operator has the behaviour of the “Operators applicable on one Scalar Value or Data Set or Data Set  
3747 Component” (see the section “Typical behaviours of the ML Operators”).

3748  
3749 *Examples*  
3750 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

3752  
3753  
3754 *Example 1:* DS\_r := sqrt(DS\_1) results in:  
3755

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

3756  
3757  
3758 *Example 2 (on components):* DS\_r := DS\_1 [ calc Me\_1 := sqrt ( Me\_1 ) ] results in:  
3759

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

3760  
3761  
3762

3763

## VTL-ML - Comparison operators

3764

Equal to : =

3765

3766

### Syntax

3767

left = right

3768

3769

### Input parameters

3770

left the left operand

3771

right the right operand

3772

3773

### Examples of valid syntaxes

3774

DS\_1 = DS\_2

3775

3776

### Semantics for scalar operations

3777

The operator returns TRUE if the left is equal to right, FALSE otherwise.

3778

For example:

3779

5 = 9 gives: FALSE

3780

5 = 5 gives: TRUE

3781

"hello" = "hi" gives: FALSE

3782

3783

### Input parameters type

3784

left,

3785

right :: dataset {measure<scalar> \_}

3786

| component<scalar>

3787

| scalar

3788

3789

### Result type

3790

result :: dataset { measure<boolean> bool\_var }

3791

| component<boolean>

3792

| boolean

3793

3794

### Additional constraints

3795

Operands left and right must be of the same scalar type

3796

3797

### Behaviour

3798

The operator has the typical behaviour of the "Operators changing the data type" (see the section "Typical

3799

behaviours of the ML Operators").

3800

3801

### Examples

3802

Given the operand Data Set DS\_1:

3803

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

3804

3805 Example 1: DS\_r := DS\_1 = 0.08 results in:  
3806

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

3807  
3808 Example 2 (on Components): DS\_r := DS\_1 [ calc Me\_2 := Me\_1 = 0.08 ]  
3809

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

3810

3811 Not equal to : <>

3812  
3813 Syntax  
3814 left <> right

3815 Input parameters  
3816 left the left operand  
3817 right the right operand

3818 Examples of valid syntaxes  
3819 DS\_1 <> DS\_2

3820 Semantics for scalar operations  
3821 The operator returns FALSE if the left is equal to right, TRUE otherwise.  
3822 For example:

3826 5 <> 9 gives: TRUE  
3827 5 <> 5 gives: FALSE  
3828 "hello" <> "hi" gives: TRUE

3829 Input parameters type  
3830 left,  
3831 right :: dataset {measure<scalar> \_}  
3832 | component<scalar>  
3833 | scalar  
3834  
3835

3836 *Result type*  
3837 result :: dataset { measure<boolean> bool\_var }  
3838 | component<boolean>  
3839 | boolean

3841 *Additional constraints*  
3842 Operands left and right must be of the same scalar type

3844 *Behaviour*  
3845 The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical  
3846 behaviours of the ML Operators”).

3848 *Examples*  
3849 Given the operand Data Sets DS\_1 and DS\_2:  
3850

DS_1				
Id_1	Id_2	Id_3	Id_4	Me_1
G	Total	Percentage	Total	7.1
R	Total	Percentage	Total	NULL

3851  
3852

DS_2				
Id_1	Id_2	Id_3	Id_4	Me_1
G	Total	Percentage	Total	7.5
R	Total	Percentage	Total	3

3853  
3854 *Example 1:* DS\_r := DS\_1 <> DS\_2 results in:  
3855

DS_r				
Id_1	Id_2	Id_3	Id_4	bool_var
G	Total	Percentage	Total	TRUE
R	Total	Percentage	Total	NULL

3856  
3857 Note that due to the behaviour for NULL values, if the value for Greece in the second operand had also been  
3858 NULL, then the result would still be NULL for Greece.

3859  
3860 *Example 2 (on Components):* DS\_r := DS\_1 [ calc Me\_2 := Me\_1<>7.5 ] results in:  
3861

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

3862  
3863

3864 Greater than : > >=

3865 *Syntax*  
3866 left { > | >= }<sup>1</sup> right  
3867

3868 *Input parameters*  
3869 left the left operand part of the comparison  
3870 right the right operand part of the comparison  
3871

3872 *Examples of valid syntaxes*  
3873 DS\_1 > DS\_2  
3874 DS\_1 >= DS\_2  
3875

3876 *Semantics for scalar operations*  
3877 The operator > returns TRUE if left is greater than right, FALSE otherwise.  
3878 The operator >= returns TRUE if left is greater than or equal to right, FALSE otherwise.  
3879 For example:  
3880 5 > 9 gives: FALSE  
3881 5 >= 5 gives: TRUE  
3882 "hello" > "hi" gives: FALSE  
3883

3884 *Input parameters type*  
3885 left,  
3886 right :: dataset {measure<scalar> \_ }  
3887 | component<scalar>  
3888 | scalar  
3889

3890 *Result type*  
3891 result :: dataset { measure<boolean> bool\_var }  
3892 | component<boolean>  
3893 | boolean  
3894

3895 *Additional constraints*  
3896 Operands left and right must be of the same scalar type  
3897

3898 *Behaviour*  
3899 The operator has the typical behaviour of the "Operators changing the data type" (see the section "Typical behaviours of the ML Operators").  
3900

3901 *Examples*  
3902 Given the operand Data Set DS\_1:  
3903  
3904

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

3905  
3906 *Example 1:* DS\_r := DS\_1 > 20 results in:  
3907

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

3908  
3909 *Example 2 (on Components):* DS\_r := DS\_1 [ calc Me\_2 := Me\_1 > 20 ] results in:  
3910

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 { < | <= }<sup>1</sup> 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.

3940 For example:  
3941 5 < 4 gives: FALSE  
3942 5 <= 5 gives: TRUE  
3943 "hello" < "hi" gives: TRUE  
3944

3945 *Input parameters type*  
3946 left, right :: dataset {measure<scalar> \_}  
3947 | component<scalar>  
3948 | scalar  
3949

3950 *Result type*  
3951 result :: dataset { measure<boolean> bool\_var }  
3952 | component<boolean>  
3953 | boolean  
3954

3955 *Additional constraints*  
3956 Operands left and right must be of the same scalar type  
3957

3958 *Behaviour*  
3959 The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical  
3960 behaviours of the ML Operators”).  
3961

3962 *Examples*  
3963 Given the operand Data Set DS\_1:  
3964

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

3965  
3966 *Example 1:* DS\_r := DS\_1 < 15000000 results in:  
3967

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

3968

3969 **Between :** **between**

3970  
3971 *Syntax*  
3972 **between** (op, from, to)  
3973



3974 *Input parameters*  
3975 op     the Data Set to be checked  
3976 from    the left delimiter  
3977 to       the right delimiter

3979 *Examples of valid syntaxes*  
3980 ds2 := between(ds1, 5,10)  
3981 ds2 := ds1 [ calc m1 := between(me2, 5, 10) ]

3983 *Semantics for scalar operations*  
3984 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:

3986           op >= from and op <= to

3988 The types of op, from and to must be compatible scalar types.

3991 *Input parameters type*  
3992 op ::           dataset {measure<scalar> \_}  
3993                | component<scalar>  
3994                | scalar  
3995  
3996 from ::         scalar | component<scalar>  
3997 to ::           scalar | component<scalar>

3999 *Result type*  
4000 result ::       dataset { measure<boolean> bool\_var }  
4001                | component<boolean>  
4002                | boolean

4004 *Additional constraints*  
4005 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.

4007 *Behaviour*  
4008 The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

4012 *Examples*

4014 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

4016  
4017 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

4019

## Element of:                    in / not\_in

### Syntax

op **in** collection  
op **not\_in** collection

collection ::= set | valueDomainName

### Input parameters

op                    the operand to be tested  
collection           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 { 0, 3, 6, 12 } 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 { 0, 3, 6, 12 } ] 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 previous 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<boolean> bool\_var }

| component<boolean>

| boolean

### Additional constraints

If op is a Data Set then it has exactly one measure.

4132 pattern is a POSIX regular expression.

4133

4134 *Behaviour*

4135 The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical  
4136 behaviours of the ML Operators”).

4137

4138 *Examples*

4139 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

4140

4141

4142 DS\_r:=(ds1, “[:alpha:]{2}[:digit:]{3}”) results in:

4143

DS_r				
Id_1	Id_2	Id_3	Id_4	bool_var
G	Total	Percentage	Total	TRUE
R	Total	Percentage	Total	FALSE

4144

4145

4146 **IsNull: isnull**

4147 *Syntax*

4148 **isnull ( op )**

4149

4150 *Input parameters*

4151 operand            mandatory            the operand

4152

4153 *Examples of valid syntaxes*

4154 isnull(DS\_1)

4155

4156 *Semantics for scalar operations*

4157 The operator returns TRUE if the value of the operand is NULL, FALSE otherwise.

4158

4159 *Examples*

4160 isnull(“Hello”) gives: FALSE

4161 isnull(NULL) gives: TRUE

4162

4163 *Input parameters type*

4164 op ::                dataset {measure<scalar> \_}

4165                      | component<scalar>

4166                      | scalar

4167

4168 *Result type*

4169 result ::            dataset { measure<boolean> bool\_var }

4170                      | component<boolean>

4171                      | boolean

4172

4173 *Additional constraints*

4174 If op is a Data Set then it has exactly one measure.

4175

4176 *Behaviour*

4177 The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical  
4178 behaviours of the ML Operators”).

4179  
4180 *Examples*

4181 Given the operand Data Set DS\_1:  
4182

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

4183  
4184 *Example 1:*      DS\_r := isnull(DS\_1)                      results in:  
4185

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

4186  
4187 *Example 2 (on Components):*      DS\_r := DS\_1[ calc Me\_2 := isnull(Me\_1) ]                      results in:  
4188

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

4189  
4190

4191 **Exists in :**      **exists\_in**

4192  
4193 *Syntax*  
4194      **exists\_in ( op1, op2 { , retain } )**  
4195  
4196      **retain ::= true | false | all**

4197  
4198 *Input parameters*

4199 op1 the operand dataset  
 4200 op2 the operand dataset  
 4201 retain the optional parameter to specify the Data Points to be returned (default: **all**)

4202

#### 4203 *Examples of valid syntaxes*

4204 exists\_in ( DS\_1, DS\_2, true )

4205 exists\_in ( DS\_1, DS\_2 )

4206 exists\_in ( DS\_1, DS\_2, all )

4207

#### 4208 *Semantics for scalar operations*

4209 This operator cannot be applied to scalar values.

4210

#### 4211 *Input parameters type*

4212 op1,

4213 op2 :: dataset

4214

#### 4215 *Result type*

4216 result :: dataset { measure<boolean> bool\_var }

4217

#### 4218 *Additional constraints*

4219 op1 has at least all the identifier components of op2 or op2 has at least all the identifier components of op1.

4220

#### 4221 *Behaviour*

4222 The operator takes under consideration the common Identifiers of op1 and op2 and checks if the combinations of values of these Identifiers which are in op1 also exist in op2.

4224 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 common Identifier Components in op1 is found in a Data Point of op2, FALSE otherwise.

4227 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.

4230 The operator has the typical behaviour of the “Operators changing the data type” (see the section “Typical behaviours of the ML Operators”).

4232

#### 4233 *Examples*

4234 Given the operand Data Sets DS\_1 and DS\_2:

4235

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

4236

4237

4238

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

4239  
4240  
4241

*Example 1:*       $DS\_r := \text{exists\_in}(DS\_1, DS\_2, \text{all})$       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

4242  
4243  
4244

*Example 2:*       $DS\_r := \text{exists\_in}(DS\_1, DS\_2, \text{true})$       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

4245  
4246  
4247

*Example 3:*       $DS\_r := \text{exists\_in}(DS\_1, DS\_2, \text{false})$       results in:

DS_r				
Id_1	Id_2	Id_3	Id_4	bool_var
2012	G	Total	Total	FALSE
2012	M	Total	Total	FALSE

4248



4249

## VTL-ML - Boolean operators

4250

Logical conjunction: **and**

4251

4252

### Syntax

4253

op1 **and** op2

4254

4255

### Input parameters

4256

op1 the first operand

4257

op2 the second operand

4258

4259

### Examples of valid syntaxes

4260

DS\_1 and DS\_2

4261

4262

### Semantics for scalar operations

4263

The **and** operator returns TRUE if both operands are TRUE, otherwise FALSE. The two operands must be of *boolean* type.

4264

For example:

4266

FALSE and FALSE gives FALSE

4267

FALSE and TRUE gives FALSE

4268

FALSE and NULL gives FALSE

4269

TRUE and FALSE gives FALSE

4270

TRUE and TRUE gives TRUE

4271

TRUE and NULL gives NULL

4272

NULL and NULL gives NULL

4273

4274

### Input parameters type

4275

op1,

4276

op2 :: dataset {measure<boolean> \_}

4277

| component<boolean>

4278

| boolean

4279

4280

### Result type

4281

result :: dataset { measure<boolean> \_}

4282

| component<boolean>

4283

| boolean

4284

4285

### Additional constraints

4286

None.

4287

4288

### Behaviour

4289

The operator has the typical behaviour of the “Behaviour of Boolean operators” (see the section “Typical behaviours of the ML Operators”).

4290

4291

4292

### Examples

4293

Given the operand Data Sets DS\_1 and DS\_2:

4294

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 [ calc 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
FALSE or NULL	gives NULL
TRUE or FALSE	gives TRUE
TRUE or TRUE	gives TRUE
TRUE or NULL	gives TRUE
NULL or NULL	gives NULL

*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:

4351

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

4352

4353

4354

Example 2 (on Components): DS\_r:= DS\_1 [ calc 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

4355

4356

## Exclusive disjunction : xor

4357

### Syntax

4358

op1 xor op2

4359

4360

### Input parameters

4361

op1 the first operand

4362

op2 the second operand

4363

4364

4365

### Examples of valid syntaxes

4366

DS\_1 xor DS\_2

4367

4368

### Semantics for scalar operations

4369

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.

4370

4371

For example:

4372

FALSE xor FALSE gives FALSE

4373

FALSE xor TRUE gives TRUE

4374

FALSE xor NULL gives NULL

4375

TRUE xor FALSE gives TRUE

4376

TRUE xor TRUE gives FALSE

4377

TRUE xor NULL gives NULL

4378

NULL xor NULL gives NULL

4379

4380

### Input parameters type

4381

op1,

4382

op2 :: dataset {measure<boolean> \_ }

4383

| component<boolean>

4384

| 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 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 [ calc 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

4409

4410 Logical negation : **not**

4411

4412

*Syntax*

4413

**not** op

4414

4415

*Input parameters*

4416

op the operand

4417

4418

*Examples of valid syntaxes*

4419

not DS\_1

4420

4421

*Semantics for scalar operations*

4422

The **not** operator returns TRUE if op is FALSE, otherwise TRUE. The input operand must be of *boolean* type.

4423

For example:

4424

not FALSE gives TRUE

4425

not TRUE gives FALSE

4426

not NULL gives NULL

4427

4428

*Input parameters type*

4429

op :: dataset {measure<boolean> \_ }

4430

| component<boolean>

4431

| boolean

4432

4433

*Result type*

4434

result :: dataset { measure<boolean> \_ }

4435

| component<boolean>

4436

| boolean

4437

4438

*Additional constraints*

4439

None.

4440

4441

*Behaviour*

4442

The operator has the typical behaviour of the “Behaviour of Boolean operators” (see the section “Typical

4443

behaviours of the ML Operators”).

4444

4445

*Examples*

4446

Given the operand Data Set DS\_1:

4447

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

4448

4449

4450

*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

4451

4452

4453

*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

4454

## 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* and may have other Identifiers.

*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 output 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

As described in the User Manual, the *time* data type is the intervening time between two time points and using the ISO 8601 standard it can be expressed through a start date and an end date separated by a slash at any precision. In the examples relevant to the *time* data type the precision is set at the level of month and the time format YYYY-MM/YYYY-MM is used.

Given the Data Set `DS_1`, which contains *annual* 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!"

4594  
4595  
4596

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!"

4597  
4598  
4599  
4600

Given the Data Set DS\_2, which contains *annual* 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!"

4601  
4602  
4603

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!"

4604  
4605  
4606

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!"

4607  
4608  
4609  
4610  
4611

Given the Data Set DS\_3, which contains *annual* time series, where Id\_2 is the reference time Identifier of *time\_period* type:

DS_3		
Id_1	Id_2	Me_1
A	2010	"hello world"
A	2012	"say hello"
A	2013	"he"
B	2011	"hi, hello! "
B	2012	"hi"
B	2014	"hello!"

4612  
4613  
4614

Example 5: DS\_r := fill\_time\_series ( DS\_3, single )

results in:

DS_r		
Id_1	Id_2	Me_1
A	2010	"hello world"
A	2011	NULL
A	2012	"say hello"
A	2013	"he"
B	2011	"hi, hello! "

B	2012	"hi"
B	2013	NULL
B	2014	"hello!"

4615  
4616  
4617

Example 6: DS\_r := fill\_time\_series ( DS\_3, all )

results in:

DS_r		
Id_1	Id_2	Me_1
A	2010	"hello world"
A	2011	NULL
A	2012	"say hello"
A	2013	"he"
A	2014	NULL
B	2010	NULL
B	2011	"hi, hello! "
B	2012	"hi"
B	2013	NULL
B	2014	"hello!"

4618  
4619  
4620  
4621  
4622

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	2010	"hello world"
A	2012	"say hello"
A	2010Q1	"he"
A	2010Q2	"hi, hello! "
A	2010Q4	"hi"
A	2011Q2	"hello!"

4623  
4624  
4625

Example 7: DS\_r := fill\_time\_series ( DS\_4, single )

results in:

DS_r		
Id_1	Id_2	Me_1
A	2010	"hello world"
A	2011	NULL
A	2012	"say hello"
A	2010Q1	"he"
A	2010Q2	"hi, hello! "
A	2010Q3	NULL
A	2010Q4	"hi"
A	2011Q2	"hello!"

4626  
4627  
4628

*Example 8:*      DS\_r := fill\_time\_series ( DS\_4, all )

results in:

DS_r		
Id_1	Id_2	Me_1
A	2010	"hello world"
A	2011	NULL
A	2012	"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

4629

4630

4631 Flow to stock : **flow\_to\_stock**

4632

4633 *Syntax*

4634 **flow\_to\_stock** ( op )

4635

4636 *Input Parameters*

4637 op the operand

4638

4639 *Examples of valid syntaxes*

4640 flow\_to\_stock ( ds\_1 )

4641

4642 *Semantics for scalar operations*

4643 This operator does not perform scalar operations.

4644

4645 *Input parameters type:*

4646 op :: dataset { identifier < time > \_ , identifier \_\* , measure<number> \_+ }

4647

4648 *Result type:*

4649 result :: dataset { identifier < time > \_ , identifier \_\* , measure<number> \_+ }

4650

4651 *Additional constraints*

4652 The operand dataset has an Identifier of type *time*, *date* or *time\_period* and may have other Identifiers.

4653

4654 *Behaviour*

4655 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”.

4656 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”.

4657 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.

4663 The flow\_to\_stock operator can be applied only on Data Sets of time series and returns a Data Set of time series.

4664 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.

4667 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.

4669

4670

4671 *Examples*

4672

4673 As described in the User Manual, the *time* data type is the intervening time between two time points and using the ISO 8601 standard it can be expressed through a start date and an end date separated by a slash at any precision. In the examples relevant to the *time* data type the precision is set at the level of month and the time format YYYY-MM/YYYY-MM is used.

4677

4678 Given the Data Set DS\_1, which contains *annual* time series, where Id\_2 is the reference time Identifier of *time* type:

4679

4680

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

4681  
4682  
4683

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

4684  
4685  
4686  
4687  
4688

Given the Data Set DS\_2, which contains *annual* 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

4689  
4690  
4691

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

4692

4693

4694

4695

Given the Data Set DS\_3, which contains *annual* time series, where Id\_2 is the reference time Identifier of *time\_period* type:

DS_3		
Id_1	Id_2	Me_1
A	2010	2
A	2011	5
A	2012	-3
A	2013	9
B	2010	4
B	2011	-8
B	2012	0
B	2013	6

4696

4697

4698

Example 3: DS\_r := flow\_to\_stock ( DS\_3 ) results in:

DS_r		
Id_1	Id_2	Me_1
A	2010	2
A	2011	7
A	2012	4
A	2013	13
B	2010	4
B	2011	-4
B	2012	-4
B	2013	2

4699

4700

4701

4702

4703

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	2010	2
A	2011	7
A	2012	4
A	2013	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	2010	2
A	2011	9
A	2012	13
A	2013	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

As described in the User Manual, the *time* data type is the intervening time between two time points and using the ISO 8601 standard it can be expressed through a start date and an end date separated by a slash at any precision. In the examples relevant to the *time* data type the precision is set at the level of month and the time format YYYY-MM/YYYY-MM is used.

Given the Data Set DS\_1, which contains *annual* 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 *annual* 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

4768

4769

4770

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

4771

4772

4773

4774

4775

Given the Data Set DS\_3, which contains *annual* time series, where Id\_2 is the reference time Identifier of *time\_period* type:

DS_3		
Id_1	Id_2	Me_1
A	2010	2
A	2011	7
A	2012	4
A	2013	13
B	2010	4
B	2011	-4
B	2012	-4
B	2013	2

4776

4777

4778

Example 3: DS\_r := stock\_to\_flow ( DS\_3 ) results in:

DS_r		
Id_1	Id_2	Me_1
A	2010	2
A	2011	5
A	2012	-3
A	2013	9
B	2010	4
B	2011	-8

B	2012	0
B	2013	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	2010	2
A	2011	9
A	2012	13
A	2013	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	2010	2
A	2011	7
A	2012	4
A	2013	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, 1 )

### Semantics for scalar operations

This operator does not perform scalar operations.

4804 *Input parameters type:*  
 4805 op :: dataset { identifier < time > \_ , identifier \_ \* }  
 4806 shiftNumber :: integer

4808 *Result type:*  
 4809 result :: dataset { identifier < time > \_ , identifier \_ \* }

4811 *Additional constraints*  
 4812 The operand dataset has an Identifier of type *time*, *date* or *time\_period* and may have other Identifiers.

4814 *Behaviour*  
 4815 This operator takes in input a Data Set of time series and, for each time series of the Data Set, shifts the reference  
 4816 time Identifier of a number of periods (of the time series) equal to the *shift\_number* parameter. If *shift\_number*  
 4817 is negative, the shift is in the past, otherwise in the future. For example, if the period of the time series is month  
 4818 and *shift\_number* is -1 the reference time Identifier is shifted of two months in the past.  
 4819 The operator can be applied only on Data Sets of time series and returns a Data Set of time series.  
 4820 The result Data Set has the same Identifier, Measure and Attribute Components as the operand Data Set and  
 4821 contains the same time series as the operand, because the time series Identifiers (all the Identifiers except the  
 4822 reference time Identifier) are not changed.  
 4823 The Attribute propagation rule is not applied.  
 4824 As mentioned in the section “Behaviour of the Time Operators”, the operator is assumed to know which is the  
 4825 *time* Identifier as well as the *period* of each data point.

4827 *Examples*  
 4828 As described in the User Manual, the *time* data type is the intervening time between two time points and using the  
 4829 ISO 8601 standard it can be expressed through a start date and an end date separated by a slash at any precision. In  
 4830 the examples relevant to the *time* data type the precision is set at the level of month and the time format YYYY-  
 4831 MM/YYYY-MM is used.

4832 Given the Data Set DS\_1, which contains *yearly* time series, where Id\_2 is the reference time Identifier of *time*  
 4833 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!"

4836  
 4837 *Example 1:* DS\_r := timeshift ( DS\_1 , -1 ) results in:  
 4838

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 *annual* 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 := timeshift ( 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 *annual* time series, where Id\_2 is the reference time Identifier of *time\_period* type:

DS_3		
Id_1	Id_2	Me_1
A	2010	"hello world"
A	2011	NULL
A	2012	"say hello"
A	2013	"he"
B	2010	"hi, hello! "
B	2011	"hi"
B	2012	NULL

B	2013	"hello!"
---	------	----------

4852  
4853  
4854

Example 3: DS\_r := timeshift ( DS\_3 , 1 ) results in:

DS_r		
Id_1	Id_2	Me_1
A	2011	"hello world"
A	2012	NULL
A	2013	"say hello"
A	2014	"he"
B	2011	"hi, hello! "
B	2012	"hi"
B	2013	NULL
B	2014	"hello!"

4855  
4856  
4857  
4858  
4859

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	2010	"hello world"
A	2011	NULL
A	2012	"say hello"
A	2013	"he"
A	2010Q1	"hi, hello! "
A	2010Q2	"hi"
A	2010Q3	NULL
A	2010Q4	"hello!"

4860  
4861  
4862

Example 4: DS\_r := time\_shift ( DS\_3 , -1 ) results in:

DS_r		
Id_1	Id_2	Me_1
A	2009	"hello world"
A	2010	NULL
A	2011	"say hello"
A	2012	"he"
A	2009Q4	"hi, hello! "
A	2010Q1	"hi"
A	2010Q2	NULL
A	2010Q3	"hello!"

4863



## 4864 Time aggregation : **time\_agg**

4865 The operator **time\_agg** converts *time*, *date* and *time\_period* values from a smaller to a larger duration.

4866  
4867 *Syntax*  
4868 **time\_agg** ( periodIndTo { , periodIndFrom } { , op } { , **first** | **last** } )  
4869

### 4870 *Input parameters*

4871 **op** the scalar value, the Component or the Data Set to be converted. If not specified, then  
4872 **time\_agg** is used in combination within an aggregation operator  
4873 **periodIndFrom** the source period indicator  
4874 **periodIndTo** the target period indicator  
4875

### 4876 *Examples of valid syntaxes*

4877 sum ( DS group all time\_agg ( Me, "A" ) )  
4878 time\_agg ( "A", cast ( "2012Q1", time\_period , "YYYY\Qq" ) )  
4879 time\_agg("M", cast ("2012-12-23", date, "YYYY-MM-DD" ) )  
4880 time\_agg("M", DS1)  
4881 ds\_2 := ds1[calc Me1 := time\_agg("M",Me1)]  
4882

### 4883 *Semantics for scalar operations*

4884 The operator converts a *time*, *date* or *time\_period* value from a smaller to a larger duration.  
4885

### 4886 *Input parameters type*

4887 **op** :: dataset { identifier < time > \_ , identifier \_\* }  
4888 | component<time>  
4889 | time  
4890 **periodIndFrom** :: duration  
4891 **periodIndTo** :: duration  
4892

### 4893 *Result type*

4894 **op** :: dataset { identifier < time > \_ , identifier \_\* }  
4895 | component<time>  
4896 | time  
4897

### 4898 *Additional constraints*

4899 If **op** is a Data Set then it has exactly an Identifier of type *time*, *date* or *time\_period* and may have other Identifiers.  
4900 It is only possible to convert smaller duration values to larger duration values (e.g. it is possible to convert  
4901 *monthly* data to *annual* data but the contrary is not allowed).  
4902

### 4903 *Behaviour*

4904 The scalar version of this operator takes as input a *time*, *date* or *time\_period* value, converts it to **periodIndTo**  
4905 and returns a scalar of the corresponding type.

4906 The Data Set version acts on a single Measure Data Set of type *time*, *date* or *time\_period* and returns a Data Set  
4907 having the same structure.

4908 Finally, VTL also provides a component version, for use in combination with an aggregation operator, because  
4909 the change of frequency requires an aggregation. In this case, the operator converts the **period\_indicator** of the  
4910 data points (e.g., convert *monthly* data to *annual* data).

4911 On *time* type, the operator maps the input value into the comprising larger regular interval, whose duration is  
4912 the one specified by the **periodIndTo** parameter.

4913 On *date* type, the operator maps the input value into the comprising larger period, whose duration is the one  
4914 specified by the **periodIndTo** parameter, which is conventionally represented either by the start or by the end  
4915 date, according to the **first/last** parameter.

4916 On *time\_period* type, the operator maps the input value into the comprising larger time period specified by the  
4917 **periodIndTo** parameter (the original period indicator is converted in the target one and the number of periods is  
4918 adjusted correspondingly).

4919 The input duration **periodIndFrom** is optional. In case of *time\_period* Data Points, the input duration can be  
4920 inferred from the internal representation of the value. In case of *time* or *date* types, it is inferred by the  
4921 implementation. Filters on input time series can be obtained with the **filter** clause.  
4922

4923  
4924  
4925  
4926

*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

4927  
4928  
4929

*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

4930  
4931  
4932  
4933  
4934  
4935  
4936  
4937  
4938  
4939  
4940  
4941  
4942  
4943  
4944  
4945  
4946  
4947

*Example 2:*      DS\_r := time\_agg ( “Q”, cast ( “2012M01”, time\_period, “YYYYMMM” ) )  
  
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”), \_ , last )  
  
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”), \_ , first )  
  
and produces a *date* value corresponding to the *string* “20120101”.

4948      Actual time :      current\_date

4949  
4950  
4951  
4952  
4953  
4954  
4955  
4956

*Syntax*  
  
current\_date ( )  
  
*Input parameters*  
None  
  
*Examples of valid syntax*

4957 `current_date`  
4958  
4959 *Semantics for scalar operations*  
4960 The operator **current\_date** returns the current time as a *date* type.  
4961  
4962 *Input parameters type*  
4963 This operator has no input parameters.  
4964  
4965 *Result type*  
4966 `result ::`                `date`  
4967  
4968 *Additional constraints*  
4969 None.  
4970  
4971 *Behaviour*  
4972 The operator return the current date  
4973  
4974 *Examples*  
4975 `cast ( current_date, string, "YYYY.MM.DD" )`  
4976

# VTL-ML - Set operators

4978

Union: **union**

4979

4980

*Syntax*

4981

**union ( dsList )**

4982

4983

dsList ::= ds { , ds }\*

4984

4985

*Input parameters*

4986

dsList the list of Data Sets in the union

4987

4988

*Examples of valid syntaxes*

4989

union ( ds2, ds3 )

4990

4991

*Semantics for scalar operations*

4992

This operator does not perform scalar operations.

4993

4994

*Input parameters type*

4995

ds :: dataset

4996

4997

*Result type*

4998

result :: dataset

4999

5000

*Additional constraints*

5001

All the Data Sets in dsList have the same Identifier, Measure and Attribute Components.

5002

5003

*Behaviour*

5004

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.

5005

5006

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 choosing 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.

5007

5008

The operator has the typical behaviour of the “Behaviour of the Set operators” (see the section “Typical behaviours of the ML Operators”).

5009

5010

The automatic Attribute propagation is not applied.

5011

5012

*Examples*

5013

5014

Given the operand Data Sets DS\_1 and DS\_2:

5015

5016

5017

5018

5019

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

5020

5021

DS_2				
Id_1	Id_2	Id_3	Id_4	Me_1

2012	N	Total	Total	23
2012	S	Total	Total	5

5022  
5023  
5024  
5025

*Example 1:*       $DS_r := \text{union}(DS_1, DS_2)$       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

5026  
5027  
5028

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

5029  
5030

DS_2				
Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	23
2012	S	Total	Total	5

5031  
5032  
5033  
5034

*Example 2:*       $DS_r := \text{union} ( DS_1, DS_2 )$       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

5035

## Intersection :                      intersect

5036  
5037  
5038  
5039  
5040  
5041  
5042  
5043

### *Syntax*

**intersect ( dsList )**

dsList ::= ds { , ds }\*

### *Input parameters*

dsList                      the list of Data Sets in the intersection

5044 *Examples of valid syntaxes*

5045 intersect ( ds2, ds3 )

5046

5047 *Semantics for scalar operations*

5048 This operator cannot be applied to scalar values.

5049

5050 *Input parameters type*

5051 ds :: dataset

5052

5053 *Return type*

5054 result :: dataset

5055

5056 *Additional constraints*

5057 All the Data Sets in dsList have the same Identifier, Measure and Attribute Components.

5058

5059 *Behaviour*

5060 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.

5063 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 choosing 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.

5068 The operator has the typical behaviour of the "Behaviour of the Set operators" (see the section "Typical behaviours of the ML Operators").

5070 The automatic Attribute propagation is not applied.

5071

5072 *Examples*

5073 Given the operand Data Sets DS\_1 and DS\_2:

5074

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

5075

5076

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

5077

5078 Example 1: DS\_r := intersect(DS\_1,DS\_2) results in:

5079

DS_1				
Id_1	Id_2	Id_3	Id_4	Me_1
2012	G	Total	Total	2

5080

5081 Set difference : **setdiff**

5082  
5083  
5084  
5085  
5086  
5087  
5088  
5089  
5090  
5091  
5092  
5093  
5094  
5095  
5096  
5097  
5098  
5099  
5100  
5101  
5102  
5103  
5104  
5105  
5106  
5107  
5108  
5109  
5110  
5111  
5112  
5113  
5114  
5115  
5116  
5117  
5118

### 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

5119  
5120  
5121

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

5122  
5123  
5124

*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

5125  
5126  
5127

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

5128  
5129

DS_2			
Id_1	Id_2	Id_3	Me_1
R	M	2011	7
R	F	2011	10

5130  
5131  
5132

*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

5133  
5134

5135

**Simmetric difference :**      **symdiff**

5136  
5137

*Syntax*

**symdiff ( ds1, ds2 )**

5138  
5139  
5140

*Input parameters*

ds1      the first Data Set in the difference  
ds2      the second Data Set in the difference

5143  
5144

*Examples of valid syntaxes*

symdiff (ds\_2, ds\_3)

5146  
5147

*Semantics for scalar operations*

This operator cannot be applied to scalar values.

5149  
5150

*Input parameters type*

ds1, ds2 ::      dataset

5152  
5153

*Result type*



5154 result :: dataset

5155

5156 *Additional constraints*

5157 The operand Data Sets have the same Identifier, Measure and Attribute Components.

5158

5159 *Behaviour*

5160 The operator implements the symmetric set difference between functions (i.e. Data Sets), interpreting the Data  
5161 Points of the input Data Sets as the elements in the operand Sets. The operator returns one Data Set, with the  
5162 same Identifier, Measure and Attribute Components as the operand Data Sets, containing the Data Points that  
5163 appear in the first Data Set but not in the second and the Data Points that appear in the second Data Set but not  
5164 in the first one.

5165 Data Points are compared to one another by Identifier Components. For symdiff (ds1, ds2), the resulting Data  
5166 Set contains all the Data Points dp1 contained in ds1 for which there is no Data Point dp2 in ds2 with the same  
5167 values for homonym Identifier components and all the Data Points dp2 contained in ds2 for which there is no  
5168 Data Point dp1 in ds1 with the same values for homonym Identifier Components.

5169 The operator has the typical behaviour of the “Behaviour of the Set operators” (see the section “Typical  
5170 behaviours of the ML Operators”).

5171 The automatic Attribute propagation is not applied.

5172

5173 *Examples*

5174 Given the operand Data Sets DS\_1 and DS\_2 :

5175

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

5176

5177

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

5178

5179 *Example 1:* DS\_r := symdiff ( DS\_1, DS\_2 ) results in:

5180

DS_r				
Id_1	Id_2	Id_3	Id_4	Me_1
2012	B	Total	Total	1
2011	B	Total	Total	1

5181

5183 Hierarchical roll-up : **hierarchy**5184 *Syntax*5185 **hierarchy** ( op , hr { **condition** condComp { , condComp }\* } { **rule** ruleComp } { mode } { input } { output } )5186 mode ::= **non\_null** | **non\_zero** | **partial\_null** | **partial\_zero** | **always\_null** | **always\_zero**5187 input ::= **dataset** | **rule** | **rule\_priority**5188 output ::= **computed** | **all**

5189

5190 *Input parameters*

5191 op the operand Data Set.

5192 hr the hierarchical Ruleset to be applied.

5193 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).

5195 ruleComp ruleComp is the Identifier of op to be associated to the rule Value Domain or Variable defined in hr.

5197 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.5200 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.5202 output this parameter specifies the content of the resulting Data Set. The meaning of the possible values of the parameter is explained below.

5203

5204

5205 *Examples of valid syntaxes*

5206 hierarchy ( DS1, HR1 rule Id\_1 non\_null all )

5207 hierarchy ( DS2, HR2 condition Comp\_1, Comp\_2 rule Id\_3 non\_zero rule computed )

5208

5209 *Semantics for scalar operations*

5210 This operator cannot be applied to scalar values.

5211

5212 *Input parameters type*

5213 op :: dataset { measure&lt;number&gt; \_ }

5214 hr :: name &lt; hierarchical &gt;

5215 condComp :: name &lt; component &gt;

5216 ruleComp :: name &lt; dentifier &gt;

5217

5218 *Result type*

5219 result :: dataset {measure&lt;number&gt; \_ }

5220

5221 *Additional constraints*

5222 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.

5226 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.

5230

5231 *Behaviour*5232 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.

5235 The behaviours relevanto to the different options of the input parameters are the following.

5236 First, the parameter **input** is considered to determine the source of the Data Points used as input of the  
5237 Hierarchy. The possible options of the parameter input and the corresponding behaviours are the following:  
5238 **dataset** For each Rule of the Ruleset and for each item on the right hand side of the Rule, the operator  
5239 takes the input Data Points exclusively from the operand **op**.  
5240 **rule** For each Rule of the Ruleset and for each item on the right-hand side of the Rule:  
5241 • if the item is not defined as the result (left-hand side) of another Rule, the current Rule  
5242 takes the input Data Points from the operand **op**  
5243 • if the item is defined as the result of another Rule, the current Rule takes the input Data  
5244 Points from the computed output of such other Rule;  
5245 **rule\_priority** For each Rule of the Ruleset and for each item on the right-hand side of the Rule:  
5246 • if the item is not defined as the result (left-hand side) of another rule, the current Rule  
5247 takes the input Data Points from the operand **op**.  
5248 • if the item is defined as the result of another Rule, then:  
5249 ○ if an expected input Data Point exists in the computed output of such other Rule  
5250 and its Measure is not NULL, then the current Rule takes such Data Point;  
5251 ○ if an expected input Data Point does not exist in the computed output of such  
5252 other Rule or its measure is NULL, then the current Rule takes the Data Point  
5253 from **op** (if any) having the same values of the Identifiers;  
5254 if the parameter input is not specified then it is assumed to be **rule**.  
5255 Then the parameter **mode** is considered, to determine the behaviour for missing Data Points and for the Data  
5256 Points to be produced in the output. The possible options of the parameter **mode** and the corresponding  
5257 behaviours are the following:  
5258 **non\_null** the result Data Point is produced when its computed Measure value is not NULL (i.e., when no  
5259 Data Point corresponding to the Code Items of the right side of the rule is missing or has NULL  
5260 Measure value); in the calculation, the possible missing Data Points corresponding to the Code  
5261 Items of the right side of the rule are considered existing and having a Measure value equal to  
5262 NULL;  
5263 **non\_zero** the result Data Point is produced when its computed Measure value is not equal to 0 (zero);  
5264 the possible missing Data Points corresponding to the Code Items of the right side of the rule  
5265 are considered existing and having a Measure value equal to 0;  
5266 **partial\_null** the result Data Point is produced if at least one Data Point corresponding to the Code Items of  
5267 the right side of the rule is found (whichever is its Measure value); the possible missing Data  
5268 Points corresponding to the Code Items of the right side of the rule are considered existing and  
5269 having a NULL Measure value;  
5270 **partial\_zero** the result Data Point is produced if at least one Data Point corresponding to the Code Items of  
5271 the right side of the rule is found (whichever is its Measure value); the possible missing Data  
5272 Points corresponding to the Code Items of the right side of the rule are considered existing and  
5273 having a Measure value equal to 0 (zero);  
5274 **always\_null** the result Data Point is produced in any case; the possible missing Data Points corresponding  
5275 to the Code Items of the right side of the rule are considered existing and having have a  
5276 Measure value equal to NULL;  
5277 **always\_zero** the result Data Point is produced in any case; the possible missing Data Points corresponding  
5278 to the Code Items of the right side of the rule are considered existing and having a Measure  
5279 value equal to 0 (zero);  
5280 If the parameter **mode** is not specified, then it is assumed to be **non\_null**  
5281  
5282 The following table summarizes the behaviour of the options of the parameter “mode”  
5283

OPTION of the MODE PARAMETER:	Missing Data Points are considered:	Null Data Points are considered:	Condition for evaluating the rule	Returned Data Points
<b>Non_null</b>	NULL	NULL	If all the involved Data Points are not NULL	Only not NULL Data Points (Zeros are returned too)
<b>Non_zero</b>	Zero	NULL	If at least one of the involved Data Points is <> zero	Only not zero Data Points (NULLS are returned too)

<b>Partial_null</b>	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)
<b>Partial_zero</b>	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)
<b>Always_null</b>	NULL	NULL	Always	Data Points of any value (NULL, not NULL and zero too)
<b>Always_zero</b>	Zero	NULL	Always	Data Points of any value (NULL, not NULL and zero too)

5284

5285 Finally the parameter output is considered, to determine the content of the resulting Data Set. The possible  
5286 options of the parameter output and the corresponding behaviours are the following:

5287 **computed** the resulting Data Set contains only the set of Data Points computed according to the Ruleset  
5288 **all** the resulting Data Set contains the union between the set of Data Points “R” computed  
5289 according to the Ruleset and the set of Data Points of op that have different combinations of  
5290 values for the Identifiers. In other words, the result is the outcome of the following (virtual)  
5291 expression: union ( setdiff (op , R) , R )

5292 If the parameter output is not specified then it is assumed to be computed.

5293

5294 *Examples*

5295 Given the following hierarchical ruleset:

```
5296
5297     define hierarchical ruleset HR_1 ( valuedomain rule VD_1 ) is
5298         A = J + K + L
5299         ; B = M + N + O
5300         ; C = P + Q
5301         ; D = R + S
5302         ; E = T + U + V
5303         ; F = Y + W + Z
5304         ; G = B + C
5305         ; H = D + E
5306         ; I = D + G
5307     end hierarchical ruleset
```

5308

5309 And given the operand Data Set DS\_1 (where At\_1 is viral and the propagation rule says that the alphabetic  
5310 order prevails the NULL prevails on the alphabetic characters and the Attribute value for missing Data Points  
5311 is assumed as NULL):

5312

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
------	---	---	----

5313  
5314  
5315  
5316

*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

5317  
5318  
5319  
5320

*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

5321  
5322  
5323  
5324

*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

5325  
5326

## VTL-ML - Aggregate and Analytic operators

5328

5329

5330

5331

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
-----------------	--	----------	--------	--------

5332

## 5333 Aggregate invocation

### 5334 *Syntax*

5335 *in a Data Set expression:*

5336 aggregateOperator ( firstOperand { , additionalOperand }\* { groupingClause } )

5337 *in a Component expression within an aggr clause)*

5338 aggregateOperator ( firstOperand { , additionalOperand }\* ) { groupingClause }

5339  
5340  
5341  
5342  
5343 aggregateOperator ::= **avg** | **count** | **max** | **median** | **min** | **stddev\_pop**  
5344 | **stddev\_samp** | **sum** | **var\_pop** | **var\_samp**  
5345 groupingClause ::= { **group by** groupingId { , groupingId }\*  
5346 | **group except** groupingId { , groupingId }\*  
5347 | **group all** conversionExpr }<sup>1</sup>  
5348 { **having** havingCondition\_ }

### 5349 *Input Parameters*

5350 aggregateOperator the keyword of the aggregate operator to invoke (e.g., **avg**, **count**, **max** ...)

5351 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)

5352 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

5353 groupingClause the following alternative grouping options:

5354 **group by** the Data Points are grouped by the values of the specified Identifiers (groupingId). The Identifiers not specified are dropped in the result.

5355 **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.

5356 **group all** converts the values of an Identifier Component using conversionExpr and keeps all the resulting Identifiers.

5357 groupingId Identifier Component to be kept (in the **group by** clause) or dropped (in the **group except** clause).

5358 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.

5359 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:

5360 
$$\text{max}(\text{obs\_value}) < 1000$$

5361 while the condition  $\text{obs\_value} < 1000$  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.:

5362 
$$\text{sum} ( \text{ds group by id1 having count} ( ) \geq 10 )$$

### 5363 *Examples of valid syntaxes*

5364 **avg** ( DS\_1 )

5365 **avg** ( DS\_1 group by Id\_1, Id\_2 )

5387 avg ( DS\_1 group except Id\_1, Id\_2 )  
5388 avg ( DS\_1 group all time\_agg ( "Q" ) )

5389  
5390 *Semantics for scalar operations*

5391 The aggregate operators cannot be applied to scalar values.

5392  
5393 *Input parameters type*

5394 firstOperand :: dataset  
5395 | component  
5396 additionalOperand :: see the type of the additional parameter (if any) of the invoked  
5397 aggregateOperator. The aggregate operators and their parameters are  
5398 described in the following sections.  
5399 groupingId :: name < identifier >  
5400 conversionExpr :: identifier  
5401 havingCondition :: component<boolean>

5402  
5403 *Result type:*

5404 result :: dataset  
5405 | component

5406  
5407 *Additional constraints*

5408 The Aggregate invocation cannot be nested in other Aggregate or Analytic invocations.

5409 The aggregate operations at component level can be invoked within the **aggr** clause, both as part of a join  
5410 operator and the **aggr** operator (see the parameter **aggrExpr** of those operators).

5411 The basic scalar types of firstOperand and additionalOperand (if any) must be compliant with the specific basic  
5412 scalar types required by the invoked operator (the required basic scalar types are described in the table at the  
5413 beginning of this chapter and in the sections of the various operators below).

5414 The conversionExpr parameter applies just one conversion operator to just one Identifier belonging to the input  
5415 Data Set. The basic scalar type of the Identifier must be compatible with the basic scalar type of the conversion  
5416 operator.

5417 If the grouping clause is omitted, then all the input Data Points are aggregated in a single group and the clause  
5418 returns a Data Set that contains a single Data Point and has no Identifiers.

5419  
5420 *Behaviour*

5421 The **aggregateOperator** is applied as usual to all the measures of the firstOperand Data Set (if invoked at Data  
5422 Set level) or to the firstOperand Component of the input Data Set (if invoked at Component level). In both cases,  
5423 the operator calculates the required aggregated values for groups of Data Points of the input Data Set. The  
5424 groups of Data Points to be aggregated are specified through the groupingClause, which allows the following  
5425 alternative options.

5426  
5427 **group by** the Data Points are grouped by the values of the specified Identifiers. The Identifiers not  
5428 specified are dropped in the result.  
5429 **group except** the Data Points are grouped by the values of the Identifiers not specified in the clause. The  
5430 specified Identifiers are dropped in the result.  
5431 **group all** converts an Identifier Component using **conversionExpr** and keeps all the Identifiers.

5432  
5433 The **having** clause is used to filter groups in the result by means of an aggregate condition evaluated on the  
5434 single groups (for example the minimum number of rows in the group).

5435 If no grouping clause is specified, then all the input Data Points are aggregated in a single group and the operator  
5436 returns a Data Set that contains a single Data Point and has no Identifiers.

5437 For the invocation at Data Set level, the resulting Data Set has the same Measures as the operand. For the  
5438 invocation at Component level, the resulting Data Set has the Measures explicitly calculated (all the other  
5439 Measures are dropped because no aggregation behaviour is specified for them).

5440 For invocation at Data Set level, the Attribute propagation rule is applied. For invocation at Component level,  
5441 the Attributes calculated within the **aggr** clause are maintained in the result; for all the other Attributes that are  
5442 defined as **viral**, the Attribute propagation rule is applied (for the semantics, see the Attribute Propagation Rule  
5443 section in the User Manual).

5444 As mentioned, the Aggregate invocation at component level can be done within the **aggr** clause, both as part of a  
5445 Join operator and the **aggr** operator (see the parameter **aggrExpr** of those operators), therefore, for a better  
5446 comprehension of the behaviour at Component level, see also those operators.



5447  
5448  
5449  
5450  
5451  
5452

### 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

5453  
5454  
5455

*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

5456  
5457  
5458  
5459  
5460  
5461  
5462  
5463

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

5464  
5465  
5466

*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** }<sup>1</sup> **between** limitClause **and** limitClause

limitClause ::= { num **preceding** | num **following** | **current data point** | **unbounded preceding** | **unbounded following** }<sup>1</sup>

### 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.

5509	<u>limitClause</u>	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:
5510		
5511		
5512		• <b>unbounded preceding</b> 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)
5513		• <b>unbounded following</b> 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)
5514		• <b>current data point</b> specifies that the window starts or ends at the current Data Point.
5515		• <b>num preceding</b> specifies either the number of <b>data points</b> to consider preceding the current data point in the order given by the orderClause (when <b>data points</b> 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 <b>range</b> is specified in the windows clause).
5516		• <b>num following</b> specifies either the number of data points to consider following the current data point in the order given by the orderClause (when <b>data points</b> 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 <b>range</b> is specified in the windows clause).
5517		If the whole windowClause is omitted then the default is <b>data points between unbounded preceding and current data point</b> .
5518		
5519		
5520		
5521		
5522		
5523		
5524		
5525		
5526		
5527		
5528		
5529		
5530		
5531		
5532	identifier	an Identifier Component of the input Data Set
5533	component	a Component of the input Data Set
5534	num	a scalar <i>number</i>
5535		

#### 5536 *Examples of valid syntaxes*

5537 sum ( DS\_1 over ( partition by Id\_1 order by Id\_2 ) )  
5538 sum ( DS\_1 over ( order by Id\_2 ) )  
5539 avg ( DS\_1 over ( order by Id\_1 data points between 1 preceding and 1 following ) )  
5540 DS\_1 [ calc M1 := sum ( Me\_1 over ( order by Id\_1 ) ) ]

#### 5542 *Semantics for scalar operations*

5543 The analytic operators cannot be applied to scalar values.

#### 5545 *Input parameters type*

5546 firstOperand :: dataset  
5547 | component  
5548 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.  
5549 identifier :: name < identifier >  
5550 component :: name < component >  
5551 num :: integer

#### 5554 *Result type*

5555 result :: dataset  
5556 | component

#### 5558 *Additional constraints*

5559 The analytic invocation cannot be nested in other Aggregate or Analytic invocations.  
5560 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).  
5561 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).

5565

### 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 windowClause 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 for 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

5602 Counting the number of data points: **count**

5603 *Aggregate syntax*

5604 **count** ( dataset { groupingClause } ) *(in a Data Set expression)*

5605 **count** ( component ) { groupingClause } *(in a Component expression within an **aggr** clause)*

5606 **count** ( ) *(in an **having** clause)*

5607

5608 *Analytic syntax*

5609 **count** ( dataset **over** ( analyticClause ) ) *(in a Data Set expression)*

5610 **count** ( component **over** ( analyticClause ) ) *(in a Component expression within a **calc** clause)*

5611

5612 *Input parameters*

5613 dataset the operand Data Set  
5614 component the operand Component  
5615 groupingClause see Aggregate invocation  
5616 analyticClause see Analytic invocation

5617

5618 *Examples of valid syntaxes*

5619 See Aggregate and Analytic invocations above, at the beginning of the section.

5620

5621 *Semantics for scalar operations*

5622 This operator cannot be applied to scalar values.

5623

5624 *Input parameters type*

5625 dataset :: dataset  
5626 component :: component

5627

5628 *Result type*

5629 result :: dataset { measure<integer> int\_var }  
5630 | component<integer>

5631

5632 *Additional constraints*

5633 None.

5634

5635 *Behaviour*

5636 The operator returns the number of the input Data Points.

5637 For other details, see Aggregate and Analytic invocations.

5638

5639 *Examples*

5640 Given the Data Set DS\_1:

5641

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

5642

5643

5644

5645

Example 1:      DS\_r := count ( DS\_1 group by Id\_1 )      results in:

DS_r	
Id_1	Int_var
2011	3
2012	2

5646

5647

5648

5649

5650

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

5651

5652

Minimum value :      **min**

5653

*Aggregate syntax*

5654

**min** ( dataset { groupingClause } )

(in a Data Set expression)

5655

**min** ( component ) { groupingClause }

(in a Component expression within an **aggr** clause)

5656

5657

*Analytic syntax*

5658

**min** ( dataset **over** ( analyticClause ) )

(in a Data Set expression)

5659

**min** ( component **over** ( analyticClause ) )

(in a Component expression within a **calc** clause)

5660

5661

*Input parameters*

5662

dataset      the operand Data Set

5663

component      the operand Component

5664

groupingClause      see Aggregate invocation

5665

analyticClause      see Analytic invocation

5666

5667

*Examples of valid syntaxes*

5668

See Aggregate and Analytic invocations above, at the beginning of the section.

5669

5670

*Semantics for scalar operations*

5671

This operator cannot be applied to scalar values.

5672

5673 *Input parameters type*  
 5674 dataset :: dataset  
 5675 component :: component

5676 *Result type*  
 5677 result :: dataset  
 5678 | component

5680 *Additional constraints*  
 5681 None.

5683 *Behaviour*  
 5684 The operator returns the minimum value of the input values.  
 5685 For other details, see Aggregate and Analytic invocations.

5687 *Examples*  
 5688 Given the Data Set DS\_1:  
 5690

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

5691  
 5692 *Example 1:* DS\_r := min ( DS\_1 group by Id\_1 ) results in:  
 5693

DS_r	
Id_1	Me_1
2011	3
2012	2

5694 **Maximum value :** **max**

5695 *Aggregate syntax*  
 5696 **max** ( dataset { groupingClause } ) *(in a Data Set expression)*  
 5697 **max** ( component ) { groupingClause } *(in a Component expression within an **aggr** clause)*

5698 *Analytic syntax*  
 5699 **max** ( dataset **over** ( analyticClause ) ) *(in a Data Set expression)*  
 5700 **max** ( component **over** ( analyticClause ) ) *(in a Component expression within a **calc** clause)*

5702 *Input parameters*  
 5703 dataset the operand Data Set  
 5704 component the operand Component  
 5705 groupingClause see Aggregate invocation  
 5706 analyticClause see Analytic invocation  
 5707  
 5708

5709 *Examples of valid syntaxes*  
 5710 See Aggregate and Analytic invocations above, at the beginning of the section.

5711 *Semantics for scalar operations*  
 5712 This operator cannot be applied to scalar values.

5713 *Input parameters type*  
 5714 dataset :: dataset  
 5715 component :: component

5716 *Result type*  
 5717 result :: dataset  
 5718 | component

5719 *Additional constraints*  
 5720 None.

5721 *Behaviour*  
 5722 The operator returns the maximum of the input values.  
 5723 For other details, see Aggregate and Analytic invocations.

5724 *Examples*  
 5725 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

5733  
 5734 *Example 1:* DS\_r := max ( DS\_1 group by Id\_1 ) results in:  
 5735

DS_r	
Id_1	Me_1
2011	7
2012	4

5736 **Median value :**      **median**

5737 *Aggregate syntax*  
 5738 **median** ( dataset { groupingClause } )      *(in a Data Set expression)*  
 5739 **median** ( component ) { groupingClause }      *(in a Component expression within an **aggr** clause)*

5740 *Analytic syntax*  
 5741 **median** ( dataset **over** ( partitionClause ) )      *(in a Data Set expression)*  
 5742 **median** ( component **over** ( partitionClause ) )      *(in a Component expression within a **calc** clause)*

5744



5745 *Input parameters*  
 5746 dataset the operand Data Set  
 5747 component the operand Component  
 5748 groupingClause see Aggregate invocation  
 5749 analyticClause see Analytic invocation  
 5750  
 5751 *Examples of valid syntaxes*  
 5752 See Aggregate and Analytic invocations above, at the beginning of the section.

5753  
 5754 *Semantics for scalar operations*  
 5755 This operator cannot be applied to scalar values.

5756  
 5757 *Input parameters type*  
 5758 dataset :: dataset {measure<number>\_+}  
 5759 component :: component<number>

5760  
 5761 *Result type*  
 5762 result :: dataset { measure<number> \_+ }  
 5763 | component<number>

5764  
 5765 *Additional constraints*  
 5766 None.

5767  
 5768 *Behaviour*  
 5769 The operator returns the median value of the input values.  
 5770 For other details, see Aggregate and Analytic invocations.

5771  
 5772 *Examples*  
 5773 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

5775  
 5776  
 5777 *Example 1:* DS\_r := median ( DS\_1 group by Id\_1 ) results in:  
 5778

DS_r	
Id_1	Me_1
2011	5
2012	3

5779 **Sum :** **sum**

5780 *Aggregate syntax*  
 5781 **sum** ( dataset { groupingClause } ) *(in a Data Set expression)*  
 5782 **sum** ( component ) { groupingClause } *(in a Component expression within an **aggr** clause)*  
 5783

5784 *Analytic syntax*  
 5785 **sum** ( dataset **over** ( analyticClause ) ) *(in a Data Set expression)*  
 5786 **sum** ( component **over** ( analyticClause ) ) *(in a Component expression within a **calc** clause)*  
 5787

5788 *Input parameters*  
 5789 dataset the operand Data Set  
 5790 component the operand Component  
 5791 groupingClause see Aggregate invocation  
 5792 analyticClause see Analytic invocation  
 5793

5794 *Examples of valid syntaxes*  
 5795 See Aggregate and Analytic invocations above, at the beginning of the section.  
 5796

5797 *Semantics for scalar operations*  
 5798 This operator cannot be applied to scalar values.  
 5799

5800 *Input parameters type*  
 5801 dataset :: dataset { measure<number> \_+ }  
 5802 component :: component<number>  
 5803

5804 *Result type*  
 5805 result :: dataset { measure<number> \_+ }  
 5806 | component<number>  
 5807

5808 *Additional constraints*  
 5809 None.  
 5810

5811 *Behaviour*  
 5812 The operator returns the sum of the input values.  
 5813 For other details, see Aggregate and Analytic invocations.  
 5814

5815 *Examples*  
 5816 Given the Data Set DS\_1 :  
 5817

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

5818  
 5819 *Example 1:* DS\_r := sum ( DS\_1 group by Id\_1 ) results in:  
 5820

DS_r	
Id_1	Me_1
2011	15
2012	6

5821

5822     **Average value :**                      **avg**

5823     *Aggregate syntax*

5824         **avg** ( dataset { groupingClause } )                      *(in a Data Set expression)*

5825         **avg** ( component ) { groupingClause }                      *(in a Component expression within an **aggr** clause)*

5826

5827     *Analytic syntax*

5828         **avg** ( dataset **over** ( analyticClause ) )                      *(in a Data Set expression)*

5829         **avg** ( component **over** ( analyticClause ) )                      *(in a Component expression within a **calc** clause)*

5830

5831     *Input parameters*

5832     dataset                      the operand Data Set

5833     component                      the operand Component

5834     groupingClause                      see Aggregate invocation

5835     analyticClause                      see Analytic invocation

5836

5837     *Examples of valid syntaxes*

5838     See Aggregate and Analytic invocations above, at the beginning of the section.

5839

5840     *Semantics for scalar operations*

5841     This operator cannot be applied to scalar values.

5842

5843     *Input parameters type*

5844     dataset ::                      dataset {measure<number> \_+}

5845     component ::                      component<number>

5846

5847     *Result type*

5848     result ::                      dataset { measure<number> \_+ }

5849                                      | component<number>

5850     *Additional constraints*

5851     None.

5852

5853     *Behaviour*

5854     The operator returns the average of the input values.

5855     For other details, see Aggregate and Analytic invocations.

5856

5857     *Examples*

5858     Given the Data Set DS\_1:

5859

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

5860

5861     *Example 1:*         DS\_r := avg ( DS\_1 group by Id\_1 )

results in:

5862

DS_r	
Id_1	Me_1
2011	5

2012	3
------	---

5863

## 5864 Population standard deviation : **stddev\_pop**

5865 *Aggregate syntax*

5866 **stddev\_pop** ( dataset { groupingClause } ) *(in a Data Set expression)*

5867 **stddev\_pop** ( component ) { groupingClause } *(in a Component expression within an **aggr** clause)*

5868

5869 *Analytic syntax*

5870 **stddev\_pop** ( dataset **over** ( analyticClause ) ) *(in a Data Set expression)*

5871 **stddev\_pop** ( component **over** ( analyticClause ) ) *(in a Component expression within a **calc** clause)*

5872

5873 *Input parameters*

5874 dataset the operand Data Set

5875 component the operand Component

5876 groupingClause see Aggregate invocation

5877 analyticClause see Analytic invocation

5878

5879 *Examples of valid syntaxes*

5880 See Aggregate and Analytic invocations above, at the beginning of the section.

5881

5882 *Semantics for scalar operations*

5883 This operator cannot be applied to scalar values.

5884

5885 *Input parameters type*

5886 dataset :: dataset { measure<number> \_+ }

5887 component :: component<number>

5888

5889 *Result type*

5890 result :: dataset { measure<number> \_+ }

5891 | component<number>

5892

5893 *Additional constraints*

5894 None.

5895

5896 *Behaviour*

5897 The operator returns the “population standard deviation” of the input values.

5898 For other details, see Aggregate and Analytic invocations.

5899

5900 *Examples*

5901

5902 Given the Data Set DS\_1:

5903

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

5904

5905 *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

5947  
5948  
5949

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

5950

## 5951 Population variance : var\_pop

5952 *Aggregate syntax*

5953 **var\_pop** ( dataset { groupingClause } ) *(in a Data Set expression)*

5954 **var\_pop** ( component ) { groupingClause } *(in a Component expression within an **aggr** clause)*

5955

5956 *Analytic syntax*

5957 **var\_pop** ( dataset **over** ( analyticClause ) ) *(in a Data Set expression)*

5958 **var\_pop** ( component **over** ( analyticClause ) ) *(in a Component expression within a **calc** clause)*

5959

5960 *Input parameters*

5961 dataset the operand Data Set  
5962 component the operand Component  
5963 groupingClause see Aggregate invocation  
5964 analyticClause see Analytic invocation

5965

5966 *Examples of valid syntaxes*

5967 See Aggregate and Analytic invocations above, at the beginning of the section.

5968

5969 *Semantics for scalar operations*

5970 This operator cannot be applied to scalar values.

5971

5972 *Input parameters type*

5973 dataset :: dataset {measure<number>\_+}  
5974 component :: component<number>

5975

5976 *Result type*

5977 result :: dataset { measure<number> \_+ }  
5978 | component<number>

5979

5980 *Additional constraints*

5981 None.

5982

5983 *Behaviour*

5984 The operator returns the “population variance” of the input values.

5985 For other details, see Aggregate and Analytic invocations.

5986

5987 *Examples*

5988 Given the Data Set DS\_1 :

5989

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.



6067 When invoked at Data Set level, it returns the first value for each Measure of the input Data Set. The first value of  
6068 different Measures can result from different Data Points.  
6069 When invoked at Component level, it returns the first value of the specified Component.  
6070 For other details, see Analytic invocation.

6071 *Examples*  
6072 Given the Data Set DS\_1 :  
6073  
6074

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

6075 *Example 1:*  
6076  
6077 DS\_r := first\_value ( DS\_1 over ( partition by Id\_1, Id\_2 order by Id\_3 data points between 1 preceding and  
6078 1 following) )  
6079  
6080 results in:  
6081  
6082

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

6083

6084 Last value : **last\_value**

6085 *Syntax*  
6086 **last\_value** ( dataset **over** ( analyticClause ) ) *(in a Data Set expression)*  
6087 **last\_value** ( component **over** ( analyticClause ) ) *(in a Component expression within a **calc** clause)*

6088 *Input parameters*  
6089 dataset the operand Data Set  
6090 component the operand Component  
6091 analyticClause see Analytic invocation  
6092  
6093

*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 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 := 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

6132

6133

## Lag : lag

6134

### Syntax

6135

6136

*in a Data Set expression:*

6137

**lag** ( dataset {, offset {, defaultValue } } **over** ( { partitionClause } orderClause) )

6138

6139

*In a Component expression within a calc clause:*

6140

**lag** ( component {, offset {, defaultValue } } **over** ( { partitionClause } orderClause ) )

6141

6142

### Input parameters

6143

dataset the operand Data Set

6144

component the operand Component

6145

offset the relative position prior to the current Data Point

6146

defaultValue the value returned when the offset goes outside of the partition.

6147

partitionClause see Analytic invocation

6148

orderClause see Analytic invocation

6149

6150

### Examples of valid syntaxes

6151

See Analytic invocation above, at the beginning of the section.

6152

6153

### Semantics for scalar operations

6154

This operator cannot be applied to scalar values.

6155

6156

### Input parameters type

6157

dataset :: dataset

6158

component :: component

6159

offset :: integer [ value > 0 ]

6160

default value :: scalar

6161

6162

### Result type

6163

result :: dataset

6164

| component

6165

6166

### Additional constraints

6167

The Aggregate invocation is not allowed.

6168

The windowClause of the Analytic invocation syntax is not allowed.

6169

6170

### Behaviour

6171

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.

6172

If defaultValue is not specified then the value returned when the offset goes outside the partition is NULL.

6173

For other details, see Analytic invocation.

6174

6175

6176

### Examples

6177

Given the Data Set DS\_1 :

6178

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 :

### 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 outside the partition.
<u>partitionClause</u>	see Analytic invocation
<u>orderClause</u>	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

6210 offset :: integer [ value > 0 ]  
6211 default value :: scalar

6212 *Result type*

6213 result :: dataset  
6214 | component

6215 *Additional constraints*

6216 The Aggregate invocation is not allowed.  
6217 The windowClause of the Analytic invocation syntax is not allowed.

6218 *Behaviour*

6219 In the ordered set of Data Points of the current partition, the operator returns the value(s) taken from the Data  
6220 Point at the specified physical offset beyond the current Data Point.  
6221 If defaultValue is not specified, then the value returned when the offset goes outside the partition is NULL.  
6222 For other details, see Analytic invocation.

6223 *Examples*

6224 Given the Data Set DS\_1  
6225

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

6230 *Example 1:* DS\_r := lead ( DS\_1 , 1 over ( partition by Id\_1 , Id\_2 order by Id\_3 ) ) results in:  
6231  
6232

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

6233

6234 Rank : rank

6235 *Syntax*

6236 rank ( over ( { partitionClause } orderClause ) ) *(in a Component expression within a calc clause)*  
6237

6238 *Input parameters*  
 6239 partitionClause see Analytic invocation  
 6240 orderClause see Analytic invocation

6241  
 6242 *Examples of valid syntaxes*  
 6243 See Analytic invocation above, at the beginning of the section.

6244  
 6245 *Semantics for scalar operations*  
 6246 This operator cannot be applied to scalar values.

6247  
 6248 *Input parameters type*  
 6249 dataset :: dataset  
 6250 component :: component

6251  
 6252 *Result type*  
 6253 result :: dataset { measure<integer> int\_var }  
 6254 | component<integer>

6255  
 6256 *Additional constraints*  
 6257 The invocation at Data Set level is not allowed.  
 6258 The Aggregate invocation is not allowed.  
 6259 The windowClause of the Analytic invocation syntax is not allowed.

6260  
 6261 *Behaviour*  
 6262 The operator returns an order number (rank) for each Data Point, starting from the number 1 and following the order  
 6263 specified in the orderClause. If some Data Points are in the same order according to the specified orderClause, the  
 6264 same order number (rank) is assigned and a gap appears in the sequence of the assigned ranks (for example, if four Data  
 6265 Points have the same rank 5, the following assigned rank would be 9).  
 6266 For other details, see Analytic invocation.

6267  
 6268 *Examples*  
 6269 Given the Data Set DS\_1:  
 6270

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

6271  
 6272  
 6273 *Example 1:*

6274  
 6275 DS\_r := DS\_1 [ calc Me2 := rank ( over ( partition by Id\_1 , Id\_2 order by Me\_1 ) ) ] results in:  
 6276

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

6277

6278

Ratio to report : **ratio\_to\_report**

6279

*Syntax*

6280

**ratio\_to\_report** ( dataset **over** ( partitionClause ) ) *(in a Data Set expression)*

6281

**ratio\_to\_report** ( component **over** ( partitionClause ) ) *(in a Component expr. within a calc clause)*

6282

6283

*Input parameters*

6284

dataset the operand Data Set

6285

component the operand Component

6286

partitionClause see Analytic invocation

6287

6288

*Examples of valid syntaxes*

6289

See Analytic invocation above, at the beginning of the section.

6290

6291

*Semantics for scalar operations*

6292

This operator cannot be applied to scalar values.

6293

6294

*Input parameters type*

6295

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

6296

component :: component<number>

6297

6298

*Result type*

6299

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

6300

| component<number>

6301

6302

*Additional constraints*

6303

The Aggregate invocation is not allowed.

6304

The orderClause and windowClause of the Analytic invocation syntax are not allowed.

6305

6306

*Behaviour*

6307

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.

6308

For other details, see Analytic invocation.

6309

6310

6311

*Examples*

6312

Given the Data Set DS\_1:

6313

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



## 6319 VTL-ML - Data validation operators

### 6320 check\_datapoint

#### 6321 Syntax

6322 **check\_datapoint** ( op , dpr { **components** listComp } { output } )

6323     listComp ::=       comp { , comp }\*

6324     output ::=       **invalid** | **all** | **all\_measures**

#### 6325 Input parameters

6326 op                   the Data Set to check

6327 dpr                  the Data Point Ruleset to be used

6328 listComp           if dpr is defined on Value Domains then listComp is the list of Components of op to be  
6329                      associated (in positional order) to the conditioning Value Domains defined in dpr. If dpr is  
6330                      defined on Variables then listComp is the list of Components of op to be associated (in  
6331                      positional order) to the conditioning Variables defined in dpr (for documentation purposes).

6332 comp                 Component of op

6333 output             specifies the Data Points and the Measures of the resulting Data Set:

6334                   **invalid**       the resulting Data Set contains a Data Point for each Data Point of op and  
6335                                   each Rule in dpr that evaluates to **FALSE** on that Data Point. The resulting  
6336                                   Data Set has the Measures of op.

6337                   **all**           the resulting Data Set contains a data point for each Data Point of op and  
6338                                   each Rule in dpr. The resulting Data Set has the *boolean* Measure bool\_var.

6339                   **all\_measures** the resulting Data Set contains a Data Point for each Data Point of op and  
6340                                   each Rule in dpr. The resulting dataset has the Measures of op and the  
6341                                   *boolean* Measure bool\_var.

6342                   If not specified then output is assumed to be invalid. See the Behaviour for further details.

#### 6343 Examples of valid syntaxes

6344 check\_datapoint ( DS1, DPR invalid )

6345 check\_datapoint ( DS1, DPR all\_measures )

#### 6347 Semantics for scalar operations

6348 This operator cannot be applied to scalar values.

#### 6350 Input parameters type:

6351 op ::               dataset

6352 dpr ::             name < datapoint >

6353 comp ::            name < component >

#### 6355 Result type:

6356 result ::          dataset

#### 6358 Additional constraints

6359 If dpr is defined on Value Domains then it is mandatory to specify listComp. The Components specified in  
6360 listComp must belong to the operand op and be defined on the Value Domains specified in the signature of dpr.

6361 If dpr is defined on Variables then the Components specified in the signature of dpr must belong to the operand  
6362 op.

6363 If dpr is defined on Variables and listComp is specified then the Components specified in listComp are the same,  
6364 in the same order, as those specified in op (they are provided for documentation purposes).

6365

6366 *Behaviour*

6367 It returns a Data Set having the following Components:

- 6368 • the Identifier Components of op
- 6369 • 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 <sup>8</sup>)
- 6370 • if the output parameter is **invalid**: the original Measures of op (no *boolean* measure)
- 6371 • 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).
- 6372 • 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).
- 6373 • the Measure errorcode that contains the errorcode specified in the rule
- 6374 • the Measure errorlevel that contains the errorlevel specified in the rule

6375 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.

6383 *Examples*

6384 define datapoint ruleset dpr1 ( variable Id\_3, Me\_1 ) is  
6385     when Id\_3 = "CREDIT" then Me\_1 >= 0 errorcode "Bad credit"  
6386     ; when Id\_3 = "DEBIT" then Me\_1 >= 0 errorcode "Bad debit"  
6387 end datapoint ruleset

6388  
6389 Given the Data Set DS\_1:  
6390

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

6391  
6392 DS\_r := check\_datapoint ( DS\_1, dpr1 )           results in:  
6393

DS_r						
Id_1	Id_2	Id_3	ruleid	obs_value	errorcode	errorlevel
2011	I	DEBIT	dpr1_2	-2	Bad debit	

6394  
6395  
6396 DS\_r := check\_datapoint ( DS\_1, dpr1 all )       results in:  
6397

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		

---

<sup>8</sup> 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		

6398

## 6399 check\_hierarchy

### 6400 *Syntax*

6401 **check\_hierarchy** ( op , hr { **condition** condComp { , condComp }\* } { **rule** ruleComp }  
6402 { mode } { input } { output } )

6403 mode ::= non\_null | non\_zero | partial\_null | partial\_zero | always\_null | always\_zero

6404 input ::= dataset | dataset\_priority

6405 output ::= invalid | all | all\_measures

6406

6407

### 6408 *Input parameters*

6409 op the Data Set to be checked

6410 hr the hierarchical Ruleset to be used

6411 condComp condComp is a Component of op to be associated (in positional order) to the conditioning  
6412 Value Domains or Variables defined in hr (if any).

6413 ruleComp ruleComp is the Identifier Component of op to be associated to the rule Value Domain or  
6414 Variable defined in hr.

6415 mode this parameter specifies how to treat the possible missing Data Points corresponding to the  
6416 Code Items in the left and right sides of the rules and which Data Points are produced in  
6417 output. The meaning of the possible values of the parameter is explained below.

6418 input this parameter specifies the source of the values used as input of the comparisons. The  
6419 meaning of the possible values of the parameter is explained below.

6420 output this parameter specifies the structure and the content of the resulting dataset. The meaning of  
6421 the possible values of the parameter is explained below.

6422

### 6423 *Examples of valid syntaxes*

6424 check\_hierarchy ( DS1, HR\_2 non\_null dataset invalid )

6425 check\_hierarchy ( DS1, HR\_3 non\_zero dataset\_priority all )

6426

### 6427 *Input parameters type*

6428 op :: dataset { measure<number> \_ }

6429 hr :: name < hierarchical >

6430 condComp :: name < component >

6431 ruleComp :: name < identifier >

6432

### 6433 *Result type*

6434 result :: dataset { measure<number> \_ }

6435

### 6436 *Additional constraints*

6437 If hr is defined on Value Domains then it is mandatory to specify the condition (if any in the ruleset hr) and the  
6438 rule parameters. Moreover, the Components specified as condComp and ruleComp must belong to the operand

6439 op and must take values on the Value Domains corresponding, in positional order, to the ones specified in the  
6440 condition and rule parameter of hr.

6441 If hr is defined on Variables, the specification of condComp and ruleComp is not needed, but they can be  
6442 specified all the same if it is desired to show explicitly in the invocation which are the involved Components: in  
6443 this case, the condComp and ruleComp must be the same and in the same order as the Variables specified in in  
6444 the condition and rule signatures of hr.

## 6445 6446 6447 *Behaviour*

6448 The **check\_hierarchy** operator applies the Rules of the Ruleset hr to check the Code Items Relations between  
6450 the Code Items present in op (as for the Code Items Relations, see the User Manual - section "Generic Model for  
6451 Variables and Value Domains"). The operator checks if the relation between the left and the right member is  
6452 fulfilled, giving TRUE in positive case and FALSE in negative case.

6453  
6454 The Attribute propagation rule is applied on each group of Data Points which contributes to the same Data Point  
6455 of the result.

6456  
6457 The behaviours relevanto to the different options of the input parameters are the following.

6458 First, the parameter input is used to determine the source of the Data Points used as input of the  
6459 check\_hierarchy. The possible options of the parameter input and the corresponding behaviours are the  
6460 following:

- |      |                  |   |
|------|------------------|---|
| 6461 | dataset          | this option addresses the case where all the input Data Points of all the Rules of the Ruleset are<br>6462 expected to be taken from the input Data Set (the operand op).   |
| 6463 |                  | For each Rule of the Ruleset and for each item on the left and right sides of the Rule, the<br>6464 operator takes the input Data Points exclusively from the operand op.   |
| 6465 | dataset_priority | this option addresses the case where the input Data Points of all the Rules of the Ruleset are<br>6466 preferably taken from the input Data Set (the operand op), however if a valid Measure value<br>6467 for an expected Data Point is not found in op, the attempt is made to take it from the computed<br>6468 output of a (possible) other Rule. |
| 6469 |                  | For each Rule of the Ruleset and for each item on the left and right sides of the Rule:   |
| 6470 |                  | • if the item is not defined as the result (left side) of another Rule that applies the Code Item<br>6471 relation "is equal to" (=), the current Rule takes the input Data Points from the operand<br>6472 op.   |
| 6473 |                  | • if the item is defined as result of another Rule R that applies the Code Item relation "is<br>6474 equal to" (=), then:   |
| 6475 |                  | ○ if an expected input Data Point exists in op and its Measure is <u>not</u> NULL, then the<br>6476 current Rule takes such Data Point from op;   |
| 6477 |                  | ○ if an expected input Data Point does not exist in op or its measure is NULL, then<br>6478 the current Rule takes the Data Point (if any) that has the same Identifiers' values<br>6479 from the computed output of the other Rule R;  |

6480 if the parameter input is not specified then it is assumed to be dataset.

6481 Then the parameter mode is considered, to determine the behaviour for missing Data Points and for the Data  
6482 Points to be produced in the output. The possible options of the parameter mode and the corresponding  
6483 behaviours are the following:

- |      |              |   |
|------|--------------|---|
| 6484 | non_null     | the result Data Point is produced when all the items involved in the comparison exist and have<br>6485 not NULL Measure value (i.e., when no Data Point corresponding to the Code Items of the left<br>6486 and right sides of the rule is missing or has NULL Measure value); under this option, in<br>6487 evaluating the comparison, the possible missing Data Points corresponding to the Code Items<br>6488 of the left and right sides of the rule are considered existing and having a NULL Measure value; |
| 6489 | non_zero     | the result Data Point is produced when at least one of the items involved in the comparison<br>6490 exist and have Measure not equal to 0 (zero); the possible missing Data Points corresponding<br>6491 to the Code Items of the left and right sides of the rule are considered existing and having a<br>6492 Measure value equal to 0;   |
| 6493 | partial_null | the result Data Point is produced if at least one Data Point corresponding to the Code Items of<br>6494 the left and right sides of the rule is found (whichever is its Measure value); the possible<br>6495 missing Data Points corresponding to the Code Items of the left and right sides of the rule are<br>6496 considered existing and having a NULL Measure value;   |

6497 partial\_zero the result Data Point is produced if at least one Data Point corresponding to the Code Items of  
6498 the left and right sides of the rule is found (whichever is its Measure value); the possible  
6499 missing Data Points corresponding to the Code Items of the left and right sides of the rule are  
6500 considered existing and having a Measure value equal to 0 (zero);  
6501 always\_null the result Data Point is produced in any case; the possible missing Data Points corresponding  
6502 to the Code Items of the left and right sides of the rule are considered existing and having a  
6503 Measure value equal to NULL;  
6504 always\_zero the result Data Point is produced in any case; the possible missing Data Points corresponding  
6505 to the Code Items of the left and right sides of the rule are considered existing and having a  
6506 Measure value equal to 0 (zero);

6507 If the parameter mode is not specified, then it is assumed to be non\_null.

6508 The following table summarizes the behaviour of the options of the parameter “mode”

6509

OPTION of the MODE PARAMETER:	Missing Data Points are considered:	Null Data Points are considered:	Condition for evaluating the rule	Returned Data Points
<b>Non_null</b>	NULL	NULL	If all the involved Data Points are not NULL	Only not NULL Data Points (Zeros are returned too)
<b>Non_zero</b>	Zero	NULL	If at least one of the involved Data Points is <> zero	Only not zero Data Points (NULLS are returned too)
<b>Partial_null</b>	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)
<b>Partial_zero</b>	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)
<b>Always_null</b>	NULL	NULL	Always	Data Points of any value (NULL, not NULL and zero too)
<b>Always_zero</b>	Zero	NULL	Always	Data Points of any value (NULL, not NULL and zero too)

6510

6511 Finally the parameter output is considered, to determine the structure and content of the resulting Data Set. The  
6512 possible options of the parameter output and the corresponding behaviours are the following:

6513 all all the Data Points produced by the comparison are returned, both the valid ones (TRUE) and  
6514 the invalid ones (FALSE) besides the possible NULL ones. The result of the comparison is  
6515 returned in the *boolean* Measure bool\_var. The original Measure Component of the Data Set op  
6516 is not returned.

6517 invalid only the invalid (FALSE) Data Points produced by the comparison are returned. The result of  
6518 the comparison (*boolean* Measure bool\_var) is not returned. The original Measure Component  
6519 of the Data Set op is returned and contains the Measure values taken from the Data Points on  
6520 the left side of the rule.

6521 all\_measures all the Data Points produced by the comparison are returned, both the valid ones (TRUE) and  
6522 the invalid ones (FALSE) besides the possible NULL ones. The result of the comparison is  
6523 returned in the *boolean* Measure bool\_var. The original Measure Component of the Data Set op  
6524 is returned and contains the Measure values taken from the Data Points on the left side of the  
6525 rule.

6526 If the parameter output is not specified then it is assumed to be invalid.

6527 In conclusion, the operator returns a Data Set having the following Components:

6528

- 6529 • all the Identifier Components of op
- 6530 • 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 <sup>9</sup>)
- 6531 • 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).
- 6532 • 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
- 6533 • 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
- 6534 • 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
- 6535 • the Measure errorcode, which contains the errorcode value specified in the Rule
- 6536 • the Measure errorlevel, which contains the errorlevel value specified in the Rule

6543 Note that a generic 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.

#### 6548 Examples

6549 See also the examples in **define hierarchical ruleset**.

6550 Given the following hierarchical ruleset:

```

6551 define hierarchical ruleset HR_1 ( valuedomain rule VD_1 ) is
6552
6553     R010 :      A = J + K + L      errorlevel 5
6554     ; R020 :      B = M + N + O      errorlevel 5
6555     ; R030 :      C = P + Q      errorcode XX errorlevel 5
6556     ; R040 :      D = R + S      errorlevel 1
6557     ; R060 :      F = Y + W + Z      errorlevel 7
6558     ; R070 :      G = B + C
6559     ; R080 :      H = D + E      errorlevel 0
6560     ; R090 :      I = D + G      errorcode YY errorlevel 0
6561     ; R100 :      M >= N      errorlevel 5
6562     ; R110 :      M <= G      errorlevel 5
6563
6564 end hierarchical ruleset

```

6565 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
2010	I	14
2010	M	2

<sup>9</sup> The content of ruleid maybe personalised in the implementation

2010	N	5
2010	O	4
2010	P	7
2010	Q	-7
2010	S	3
2010	T	9
2010	U	NULL
2010	V	6

6570

6571

6572

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

6573

6574

## 6575 check

### 6576 Syntax

6577 **check** ( op { **errorcode** errorcode } { **errorlevel** errorlevel } { **imbalance** imbalance } { output } )

6578 output ::= **invalid** | **all**

### 6579 Input parameters

6580 op a *boolean* Data Set (a *boolean* condition expressed on one or more Data Sets)

6581 errorcode the error code to be produced when the condition evaluates to FALSE. It must be a valid value  
6582 of the errorcode\_vd Value Domain (or *string* if the errorcode\_vd Value Domain is not found).  
6583 It can be a Data Set or a *scalar*. If not specified then errorcode is NULL.

6584 errorlevel the error level to be produced when the condition evaluates to FALSE. It must be a valid value  
6585 of the errorlevel\_vd Value Domain (or *integer* if the errorcode\_vd Value Domain is not found).  
6586 It can be a Data Set or a *scalar*. If not specified then errorlevel is NULL.

6587 imbalance the imbalance to be computed. imbalance is a *numeric* mono-measure Data Set with the same  
6588 Identifiers of op. If not specified then imbalance is NULL.

6589 output specifies which Data Points are returned in the resulting Data Set:

6590                               **invalid**                               returns the Data Points of op for which the condition evaluates to  
6591   FALSE  
6592                               **all**                               returns all Data Points of op  
6593                               If not specified then output is **all**.

6594   *Examples of valid syntaxes*

6595   check ( DS1 > DS2 errorcode myerrorcode errorlevel myerrorlevel imbalance DS1 - DS2 invalid )

6596   *Input parameters type:*

6597   op ::               dataset  
6598   errorcode ::       errorcode\_vd  
6599   errorlevel ::       errorlevel\_vd  
6600   imbalance ::       number

6601   *Result type:*

6602   result ::           dataset

6603   *Additional constraints*

6604   op has exactly a *boolean* Measure Component.

6605   *Behaviour*

6606   It returns a Data Set having the following components:

- 6607       • the Identifier Components of op
- 6608       • a *boolean* Measure named **bool\_var** that contains the result of the evaluation of the *boolean* dataset op
- 6609       • the Measure imbalance that contains the specified imbalance
- 6610       • the Measure errorcode that contains the specified errorcode
- 6611       • the Measure errorlevel that contains the specified errorlevel

6612   If output is **all** then all data points are returned. If output is **invalid** then only the Data Points where bool\_var is  
6613   FALSE are returned.

6614   *Examples*

6615   Given the Data Sets DS\_1 and DS\_2 :

6618

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



6619

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

6620

6621 *Example 1:*      DS\_r := check ( DS1 >= DS2 imbalance DS1 - DS2 )      returns:

6622

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

6623

6625 **if-then-else :** **if**

6626

6627 *Syntax*6628 **if** condition **then** thenOperand **else** elseOperand

6629

6630 *Input parameters*

6631

6632 condition a Boolean condition (dataset, component or scalar)

6633 thenOperand the operand returned when condition evaluates to **true**6634 elseOperand the operand returned when condition evaluates to **false**

6635

6636 *Examples of valid syntaxes*

6637 if A &gt; B then A else B

6638

6639 *Semantics for scalar operations*6640 The **if** operator returns thenOperand if condition evaluates to **true**, elseOperand otherwise. For example, considering the statement:6642 **if** x1 > x2 **then** 2 **else** 5,

6643 for x1 = 3, x2 = 0 it returns 2

6644 for x1 = 0, x2 = 3 it returns 5

6645

6646 *Input Parameters type*

6647 condition :: dataset { measure &lt;boolean&gt; \_ }

6648 | component&lt;Boolean&gt;

6649 | boolean

6650 thenOperand :: dataset

6651 | component

6652 | scalar

6653 elseOperand :: dataset

6654 | component

6655 | scalar

6656

6657 *Result type*

6658 result :: dataset

6659 | component&lt;

6660 | scalar

6661

6662 *Additional constraints*

- 6663 • The operands thenOperand and elseOperand must be of the same scalar type.
- 6664 • If the operation is at scalar level, thenOperand and elseOperand are scalar then condition must be scalar too (a *boolean* scalar).
- 6665 • 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.
- 6666 • 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
  - 6667 ○ If thenOperand and elseOperand are both Data Sets then they must have the same Components in the same roles
  - 6668 ○ 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.

6669

6670

6671

6672

6673

6674

6675

6676

6677

6678

### 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

6701 DS\_r := if ( DS\_cond#Id\_4 = "F" ) then DS\_1 else DS\_2 returns:  
6702

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

6703 Nvl : nv

6704 Syntax  
6705 nv ( op1 , op2 )

6706  
6707 Input parameters  
6708 op1 the first operand  
6709 op2 the second operand

6710  
6711 Examples of valid syntaxes  
6712 nv ( ds1#m1, 0 )

6713  
6714 Semantics for scalar operations  
6715 The operator nv returns op2 when op1 is **null**, otherwise op1. For example:  
6716 nv ( 5, 0 ) returns 5  
6717 nv ( null, 0 ) returns 0

6718  
6719 Input Parameters type  
6720 op1 :: dataset  
6721 | component<scalar>  
6722 | scalar  
6723  
6724 op2 :: dataset  
6725 | component  
6726 | <scalar>

6727  
6728 Result type  
6729 result :: dataset  
6730 | component  
6731 | scalar

6732  
6733 Additional constraints  
6734 If op1 and op2 are scalar values then they must be of the same type.  
6735 If op1 and op2 are Components then they must be of the same type.  
6736 If op1 and op2 are Data Sets then they must have the same Components.

6737  
6738 Behaviour  
6739 The operator nv returns the value from op2 when the value from op1 is null, otherwise it returns the value from  
6740 op1.  
6741 The operator has the typical behaviour of the operators applicable on two scalar values or Data Sets or Data Set  
6742 Components.  
6743 Also the following statement gives the same result: if isnull ( op1 ) then op2 else op1

6744 Examples  
6745  
6746 Example 1: Given the input Data Set DS\_1  
6747  
6748

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

6749  
6750  
6751

6753 Filtering Data Points : filter

6754  
6755 *Syntax*  
6756 op [ filter filterCondition ]

6757 *Input parameters*  
6758 op the operand  
6760 filterCondition the filter condition

6761 *Examples of valid syntaxes*  
6762 DS\_1 [ filter Me\_3 > 0 ]  
6764 DS\_1 [ filter Me\_3 + Me\_2 <= 0 ]

6765 *Semantics for scalar operations*  
6766 This operator cannot be applied to scalar values.

6767 *Input parameters type:*  
6769 op :: dataset  
6771 filterCondition :: component<boolean>

6772 *Result type:*  
6773 result :: dataset

6774 *Additional constraints:*  
6775 None.

6776 *Behaviour*  
6777 The operator takes as input a Data Set (op) and a *boolean* Component expression (filterCondition) and filters the  
6780 input Data Points according to the evaluation of the condition. When the expression is TRUE the Data Point is  
6781 kept in the result, otherwise it is not kept (in other words, it filters out the Data Points of the operand Data Set  
6782 for which filterCondition condition evaluates to FALSE or NULL).

6783 *Examples*

6784  
6785 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

6788  
6789 *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** { calcRole } calcComp := calcExpr { , { calcRole } calcComp := calcExpr }\* ]

calcRole ::= **identifier** | **measure** | **attribute** | **viral attribute**

### Input parameters

op                    the operand  
calcRole            the role to be 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** aggrClause { groupingClause } ]

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

groupingClause ::= { **group by** groupingId { , groupingId }\*  
 | **group except** groupingId { , groupingId }\*  
 | **group all** conversionExpr }<sup>1</sup>  
 { **having** havingCondition }

aggrRole ::= **measure** | **attribute** | **viral attribute**

### Input Parameters

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



6875 **aggrExpr** expression at component level, having only Components of the input Data Sets as  
 6876 operands, which invokes an aggregate operator (e.g. **avg**, **count**, **max** ... , see also the  
 6877 corresponding sections) to perform the desired aggregation. Note that the **count**  
 6878 operator is used in an **aggrClause** without parameters, e.g.:

6879 DS\_1 [ aggr Me\_1 := count ( ) group by Id\_1 ) ]

6880 **groupingClause** the following alternative grouping options:

6881 **group by** the Data Points are grouped by the values of the specified Identifiers  
 6882 (groupingId). The Identifiers not specified are dropped in the result.

6883 **group except** the Data Points are grouped by the values of the Identifiers not  
 6884 specified as groupingId. The Identifiers specified as groupingId are  
 6885 dropped in the result.

6886 **group all** converts the values of an Identifier Component using conversionExpr  
 6887 and keeps all the resulting Identifiers.

6888 groupingId Identifier Component to be kept (in the **group by** clause) or dropped (in the **group**  
 6889 **except** clause).

6890 conversionExpr specifies a conversion operator (e.g., **time\_agg**) to convert an Identifier from finer to  
 6891 coarser granularity. The conversion operator is applied on an Identifier of the operand  
 6892 Data Set op.

6893 havingCondition a condition (boolean expression) at component level, having only Components of the  
 6894 input Data Sets as operands (and possibly constants), to be fulfilled by the groups of  
 6895 Data Points: only groups for which havingCondition evaluates to TRUE appear in the  
 6896 result. The havingCondition refers to the groups specified through the groupingClause,  
 6897 therefore it must invoke aggregate operators (e.g. **avg**, **count**, **max** ..., see also the  
 6898 section Aggregate invocation). A correct example of havingCondition is:

6899 max(obs\_value) < 1000

6900 instead the condition obs\_value < 1000 is not a right havingCondition, because it  
 6901 refers to the values of the single Data Points and not to the groups. The **count** operator  
 6902 is used in a havingCondition without parameters, e.g.:

6903 sum (DS\_1 group by id1 having count ( ) >= 10 )

6904  
 6905 *Examples of valid syntaxes*

6906 DS\_1 [ aggr M1 := min ( Me\_1 ) group by Id\_1, Id\_2 ]  
 6907 DS\_1 [ aggr M1 := min ( Me\_1 ) group except Id\_1, Id\_2 ]  
 6908

6909 *Semantics for scalar operations*

6910 This operator cannot be applied to scalar values.

6911  
 6912 *Input parameters type:*

6913 op :: dataset  
 6914 aggrComp :: name < component >  
 6915 aggrExpr :: component<scalar>  
 6916 groupingId :: name <identifier >  
 6917 conversionExpr :: identifier<scalar>  
 6918 havingCondition :: component<boolean>  
 6919

6920 *Result type:*

6921 result :: dataset  
 6922

6923 *Additional constraints*

6924 The aggrComp parameter cannot be the name of an Identifier component.

6925 All the components used in aggrExpr must belong to the operand Data Set op.

6926 The conversionExpr parameter applies just one conversion operator to just one Identifier belonging to the input  
 6927 Data Set. The basic scalar type of the Identifier must be compatible with the basic scalar type of the conversion  
 6928 operator.  
 6929

### 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		
------	--	--



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.

7093  
7094 *Examples*

7095  
7096 Given the Data Set DS\_1:  
7097

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

7098  
7099 *Example1:*      DS\_r := DS\_1 [ rename Me\_1 to Me\_2, At\_1 to At\_2] results in:  
7100

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

7101 **Pivoting :      pivot**

7102  
7103 *Syntax*  
7104      op [ **pivot** identifier , measure ]  
7105

7106 *Input parameters*  
7107 op                      the operand  
7108 identifier              the Identifier Component of op to pivot  
7109 measure                the Measure Component of op to pivot  
7110

7111  
7112 *Examples of valid syntaxes*  
7113 DS\_1 [ pivot Id\_2, Me\_1 ]  
7114

7115 *Semantics for scalar operations*  
7116 This operator cannot be applied to scalar values.  
7117

7118 *Input Parameters type*  
7119 op ::                    dataset  
7120 identifier ::            name < identifier >  
7121 measure ::              name < measure >  
7122

7123 *Result type*  
7124 result ::                dataset  
7125

7126 *Additional constraints*  
7127 The Measures created by the operator according to the behaviour described below must be defined on the same  
7128 Value Domain as the input Measure.  
7129

7130 *Behaviour*

7131 The operator transposes several Data Points of the operand Data Set into a single Data Point of the resulting Data  
7132 Set. The semantics of **pivot** can be procedurally described as follows.  
7133  
7134 1. It creates a virtual Data Set VDS as a copy of op  
7135 2. It drops the Identifier Component **identifier** and all the Measure Components from VDS.  
7136 3. It groups VDS by the values of the remaining Identifiers.  
7137 4. For each distinct value of **identifier** in op, it adds a corresponding measure to VDS, named as the value of  
7138 **identifier**. These Measures are initialized with the NULL value.  
7139 5. For each Data Point of op, it finds the Data Point of VDS having the same values as for the common  
7140 Identifiers and assigns the value of **measure** (taken from the current Data Point of op) to the Measure of  
7141 VDS having the same name as the value of **identifier** (taken from the Data Point of op).  
7142

7143 The result of the last step is the output of the operation.  
7144

7145 Note that **pivot** may create Measures whose names are non-regular (i.e. they may contain special characters,  
7146 reserved keywords, etc.) according to the rules about the artefact names described in the User Manual (see the  
7147 section “The artefact names” in the chapter “VTL Transformations”). As said in the User Manual, those names  
7148 must be quoted to be referenced within an expression.  
7149

7150 *Examples*

7151 Given the Data Set DS\_1:

7152

7153

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

7154

7155 *Example1:* DS\_r := Ds\_1 [ pivot Id\_2, Me\_1 ] results in:

7156

DS_r			
Id_1	A	B	C
1	5	2	7
2	3	4	9

7157

7158 Unpivoting : **unpivot**

7159

7160 *Syntax*

7161 op [ **unpivot** identifier , measure ]

7162

7163 *Input parameters*

7164 op the dataset operand  
7165 identifier the Identifier Component to be created  
7166 measure the Measure Component to be created

7167

7168 *Examples of valid syntaxes*

7169 DS [ unpivot Id\_5, Me\_3 ]  
7170  
7171 *Semantics for scalar operations*  
7172 This operator cannot be applied to *scalar* values.  
7173

7174 *Input Parameters type*  
7175 op :: dataset  
7176 identifier :: name < identifier >  
7177 measure :: name < measure >  
7178

7179 *Result type*  
7180 result :: dataset  
7181

7182 *Additional constraints*  
7183 All the measures of op must be defined on the same Value Domain.  
7184

7185 *Behaviour*  
7186 The **unpivot** operator transposes a single Data Point of the operand Data Set into several Data Points of the  
7187 result Data set. Its semantics can be procedurally described as follows.  
7188

- 7189 1. It creates a virtual Data Set VDS as a copy of op  
7190 2. It adds the Identifier Component identifier and the Measure Component measure to VDS.  
7191 3. For each Data Point DP and for each Measure M of op whose value is not NULL, the operator inserts a  
7192 Data Point into VDS whose values are assigned as specified in the following points  
7193 4. The VDS Identifiers other than identifier are assigned the same values as the corresponding Identifiers of  
7194 the op Data Point  
7195 5. The VDS identifier is assigned a value equal to the **name** of the Measure M of op  
7196 6. The VDS measure is assigned a value equal to the **value** of the Measure M of op  
7197

7198 The result of the last step is the output of the operation.  
7199

7200 When a Measure is NULL then **unpivot** does not create a Data Point for that Measure.  
7201 Note that in general pivoting and unpivoting are not exactly symmetric operations, i.e., in some cases the unpivot  
7202 operation applied to the pivoted Data Set does not recreate exactly the original Data Set (before pivoting).  
7203

7204 *Examples*  
7205

7206 Given the Data Set DS\_1:  
7207

DS_1			
Id_1	A	B	C
1	5	2	7
2	3	4	9

7208  
7209  
7210 *Example1:* DS\_r := DS\_1 [ unpivot Id\_2, Me\_1] results in:  
7211

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

7212

7213 Subspace :      **sub**

7214

7215 *Syntax*

7216            op [ **sub** identifier = value { , identifier = value }\* ]

7217

7218 *Input parameters*

7219 op                dataset

7220 identifier        Identifier Component of the input Data Set op

7221 value             valid value for identifier

7222

7223 *Examples of valid syntaxes*

7224 DS\_r := DS\_1 [ Id\_2 = "A", Id\_5 = 1 ]

7225

7226 *Semantics for scalar operations*

7227 This operator cannot be applied to scalar values.

7228

7229 *Input Parameters type*

7230 op ::              dataset

7231 identifier ::      name < identifier >

7232 value ::           scalar

7233

7234 *Result type*

7235 result ::          dataset

7236

7237 *Additional constraints*

7238 The specified Identifier Components identifier(s) must belong to the input Data Set op.

7239 Each Identifier Component can be specified only once.

7240 The specified value must be an allowed value for identifier.

7241

7242

7243 *Behaviour*

7244

7245 The operator returns a Data Set in a subspace of the one of the input Dataset. Its behaviour can be procedurally  
7246 described as follows:

7247

7248            1. It creates a virtual Data Set VDS as a copy of op

7249            2. It maintains the Data Points of VDS for which identifier = value (for all the specified identifier) and  
7250                eliminates all the Data Points for which identifier <> value (even for only one specified identifier)

7251            3. It projects out ("drops", in VTL terms) all the identifier(s)

7252

7253 The result of the last step is the output of the operation.

7254

7255 The resulting Data Set has the Identifier Components that are not specified as identifier(s) and has the same  
7256 Measure and Attribute Components of the input Data Set.

7257

7258 The result Data Set does not violate the functional constraint because after the filter of the step 2, all the  
7259 remaining identifier(s) do not contain the same Values for all the Data Points. In other words, given that the input  
7260 Data Set is a 1<sup>st</sup> order function and therefore does not contain duplicates, the result Data Set is a 1<sup>st</sup> order  
7261 function as well. To show this, let  $K_1, \dots, K_m, \dots, K_n$  be the Identifier components for the generic input Data Set DS.  
7262 Let us suppose that  $K_1, \dots, K_m$  are assigned to fixed values by using the subspace operator. A duplicate could arise  
7263 only if in the result there are two Data Points  $DP_{r1}$  and  $DP_{r2}$  having the same value for  $K_{m+1}, \dots, K_n$ , but this is  
7264 impossible since such Data Points had same  $K_1, \dots, K_m$  in the original Data Set DS, which did not contain  
7265 duplicates.

If we consider the vector space of Data Points individuated by the n-uples of Identifier components of a Data Set  $DS(K_1, \dots, K_n, \dots)$  (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

*Example 1:*  $DS_r := DS_1 [ \text{sub } Id_1 = 1, Id_2 = "A" ]$

results in:

DS_r		
Id_3	Me_1	At_1
XX	20	F
YY	1	F

*Example 2:*  $DS_r := DS_1 [ \text{sub } Id_1 = 1, Id_2 = "B", Id_3 = "YY" ]$  results in:

DS_r	
Me_1	At_1
9	F

*Example 3:*  $DS_r := DS_1 [ \text{sub } Id_2 = "A" ] + DS_1 [ \text{sub } Id_2 = "B" ]$  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

7290  
7291